

Apple IIGS Assembly Language Programming

Apple IIGS Assembly Language Programming

Leo J. Scanlon



BANTAM BOOKS

TORONTO • NEW YORK • LONDON • SYDNEY • AUCKLAND

Trademarks

Apple, Apple II, II + , IIe, IIC, IIGS, Applesoft, Disk II, ProDOS, QuickDraw, and SANE are registered trademarks, trademarks, or copyrighted by Apple Computer, Inc.

Macintosh is a trademark licensed to Apple Computer, Inc.

ORCA/M is a trademark of The Byte Works, Inc.

IBM is a trademark of International Business Machines Corporation.

Figures 6-3, 11-3, 12-1, 12-2, 12-3, 12-4, 12-6,
are used with the permission of Apple Computer, Inc.

Apple IIGS Assembly Language Programming
A Bantam Book / August 1987

All rights reserved.

Copyright © 1987 by Leo Scanlon.

Cover design copyright © 1987 by Bantam Books, Inc.

Interior design by Margaret Fletcher, Slawson Communications, Inc., San Diego, CA.

Production by Slawson Communications, Inc., San Diego, CA.

*This book may not be reproduced in whole or in part, by
mimeograph or any other means, without permission.*

For information address: Bantam Books, Inc.

ISBN 0-553-34395-5

Published simultaneously in the United States and Canada

Bantam Books are published by Bantam Books, Inc. Its trademark, consisting of the words "Bantam Books" and the portrayal of a rooster, is Registered in U.S. Patent and Trademark Office and in other countries. Marca Registrada. Bantam Books, Inc., 666 Fifth Avenue, New York, New York 10103.

PRINTED IN THE UNITED STATES OF AMERICA

B 0 9 8 7 6 5 4 3 1

Acknowledgments

While my name is the only one on the cover of this book, many people helped along the way, and I want to thank them publicly. Top honors belong to Bill Gladstone of Waterside Productions, Steve Guty of Bantam Books, and Martha Steffen of Apple Computer for their encouragement and unflagging enthusiasm for this project.

Special thanks also to Jim Merritt, Manager of Apple's Developer Technical Support department, who reviewed the manuscript for technical accuracy and overall usability. If you find any errors in the book, blame me, not Jim. (And please write to me about them, in care of Bantam Books.) Pete McDonald, Guillermo Ortiz and the rest of Jim's staff deserve credit, too, for patiently explaining some of the more subtle details of the IIGS.

A few other folks also provided help — without realizing it! I refer especially to Eagle Berns and the others at Apple who developed sample programs to aid IIGS developers. Their listings showed the “right” way to get things done and saved me untold hours (or days or weeks) of programming time.

Danny Goodman's excellent book *The Apple IIGS Toolbox Revealed* was also a valuable resource. It gave me a good starting point for digging deeper into Apple's voluminous documentation.

Finally, I thank Pat, who has always been there to listen.

Contents

Preface	vii
Chapter 0 — About Computer Numbering	1
Binary Numbering	2
Converting Decimal Values to Binary	3
Bytes	4
Adding Binary Numbers	4
Signed Numbers	5
Two's-Complement	6
Hexadecimal Numbering	7
Chapter 1 — Introduction to Assembly Language	9
What Is Assembly Language?	9
Overview of the 65816 Microprocessor	10
Data and Address Buses	10
Memory Organization	11
Software Features	11
Interrupts	12
Operating Modes	13
Internal Registers	13
General-Purpose Registers	15
Data Bank Register	15
Stack Pointer	15
Direct Register	16
Program Counter	16

Program Bank Register	16
Processor Status Register	17
Inside the Apple IIGS	19
Fast (2.5-MHz) Components	20
Slow (1-MHz) Components	20
System Memory	22
Read/Write Memory	23
Memory Shadowing	23
Banks \$E0 and \$E1	23
ROM	25
Programming the Apple IIGS	25
Chapter 2 — Using an Assembler	27
Developing an Assembly Language Program	28
Editor	29
Assembler	29
Linker	29
Debugger	29
Top-Down Program Design	31
Source Statements	32
Assembly Language Instructions	33
Label Field	33
Mnemonic (Opcode) Field	34
Operand Field	34
Comment Field	35
Assembler Directives	35
Program Control Directives	36
File Control Directives	39
Space Allocation Directives	39
Equate Directives	40
Listing Directives	41
Mode Directives	42
Advanced Directives	42
Operators	42
Arithmetic Operators	43
Logical Operators	45
Relational Operators	46
Entering, Assembling, and Running Programs	47
Starting the <i>Apple IIGS Programmer's Workshop</i>	47
A Simple Speaker-Beeeping Program	48
Entering the Program	49

Correcting Typing Errors	50
Leaving the Editor	53
Assemble, Link, and Run Commands	54
Shell Load Files and System Load Files	62
Automating the Assembly Process	62
Multisegment Programs	63
Debugger	63
Starting the Debugger	63
Subdisplays on the Debugger Screen	64
Single-Step and Trace Commands	66
Editing Commands	69
Register Commands	69
Memory Commands	70
Disassembly Commands	70
Conversion Commands	71
Breakpoints	71
Leaving the Debugger	72
Chapter 3 — 65816 Addressing Modes	73
Immediate	74
Accumulator	75
Implied	75
Absolute and Absolute Long	75
Absolute Indirect	77
Absolute Indexed with X or Y	78
Absolute Long Indexed with X	79
Absolute Indexed Indirect	79
Direct	80
Direct Indirect and Direct Indirect Long	81
Direct Indexed with X or Y	82
Direct Indirect Indexed and Direct Indirect Indexed Long	82
Direct Indexed Indirect	83
Program Counter Relative and Program Counter Relative Long	83
Stack	84
Stack Relative and Stack Relative Indirect Indexed	84
Block Move	85
Addressing Mode Summary	85
Read-Modify-Write Instructions	86
Final Thoughts on Addressing Modes	88

Chapter 4 — 65816 Instruction Set	90
Instruction Types	91
Alternate Mnemonics	91
Functional Groups	94
Data Transfer Instructions	94
Load and Store	94
Register Transfer	96
Block Move	96
Arithmetic Instructions	99
Data Formats	99
Addition	100
How the 65816 Subtracts	102
Subtraction	103
Signed Arithmetic	104
Increment and Decrement	104
Compare	105
Control Transfer Instructions	106
Unconditional Transfer	106
Conditional Transfer	107
Using Branch Instructions with Compares	109
Subroutine Instructions	111
Stack Instructions	113
Overview of the Stack	115
Push and Pull Registers	115
Push Immediate Data or Effective Address	118
Bit Manipulation Instructions	119
Logical	119
Bit Testing	122
Processor Status Bits	123
Shift and Rotate Instructions	124
Shifts	124
Rotates	125
Shifting Signed Numbers	126
Mode Control Instruction	127
Interrupt-Related Instructions	128
Interrupt Control	129
Return from Interrupt	129
Software Interrupts	130
Wait for Interrupt	130
Miscellaneous Instructions	131

Chapter 5 — Macros	132
Introduction to Macros	132
Macros Vs. Subroutines	133
Macros Speed Up Programming	133
Contents of Macros	134
Macro Directives	136
Macro Language Directives	136
Library Directives	139
Symbolic Parameter Directives	139
Branching Directives	140
Listing Directives	141
Creating Macro Libraries	142
Macros on the <i>Programmer's Workshop</i> Disk	143
ProDOS 16 Macros	144
Utility Macros	144
Push and Pull Macros	144
Load and Store Macros	149
Add and Subtract Macros	150
Define Macros	150
Move Macros	151
Shift Macros	152
Mode Macros	153
Write Macros	153
Check Error Macro	154
Using Predefined Macros	154
 Chapter 6 — The Apple IIGS Toolbox	 155
Tool Locator	156
Memory Manager	156
QuickDraw II	157
Managers	158
Window Manager	158
Menu Manager	158
Control Manager	159
Event Manager	160
Dialog Manager	161
Desk Manager	161
Sound Manager	162
Other Managers	162
Other Tool Sets	163

Line Editor	163
Text Tools	163
Integer Math Tools	163
Standard File Operations	163
SANE	163
Miscellaneous Tools	164
Tool Set Interactions	164
Using Tool Calls in Programs	165
Making Tool Calls	166
Calling Conventions	166
Words, Integers, and Pointers	168
General Structure of an Application Program	168
Using the Program Bank as the Data Bank	168
Starting the Tool Locator and Memory Manager	169
Allocating Working Space in Bank 0	169
Starting ROM-Based Tool Sets	171
Reading RAM-Based Tools from Disk	171
Starting RAM-Based Tool Sets	176
Shutting Down the Tool Sets	176
Leaving the Program	177
Generalized Program Model	177
What's In the Model	183
Creating the Model File	183
Validating MODEL.SRC	183
Using the Model	184
Copying Between Programs	186
Tool Locator, Memory Manager, and Miscellaneous Tools Calls	186
Start-Up and Shut-Down Sequences	186
Common Programming Errors	189
 Chapter 7 — Drawing with QuickDraw	 191
Graphics Modes	191
Drawing Environment	192
Conceptual Drawing Space	192
Pixels and Points	194
QuickDraw Draws with a Pen	194
Starting and Stopping QuickDraw	196
The Pen	198
Pen Location	198
Pen Size	198

Pen Mode	198
Pen Pattern	198
Pen Mask	201
Pen State Tool Calls	202
Colors	202
320 Mode Colors	205
640 Mode Colors	206
Dithering in 640 Mode	207
Tool Calls for Colors	208
Drawing Lines, Rectangles, and Polygons	209
Lines	210
Rectangles	211
A Program that Draws Rectangles	213
Polygons	214
Drawing Other Shapes	219
Ovals	220
Regions	225
Calculation Calls for Shapes	232
A Color-Dithering Program	232
Mouse Pointer Tool Call	236
Text	237
Pixel Images	237
Macintosh Bit Maps	238
Contents of a Pixel Image	241
Image Width	241
BoundsRect	242
The GrafPort	243
Entries in the GrafPort Record	244
Multiple GrafPorts	247
Displaying a Pixel Image	248
Tool Calls to Display Pixel Images	248
Pencil Display Program	250
Animation	250
Animating Shapes	255
Animating Pixel Images	255
Time and Date Operations	255
Tool Calls for Reading the Time and Date	260
Generating Delays	262
Working with Color Tables	267

Chapter 8 — Events	271
Modal Programs	271
The Event Loop	273
Event Types	273
Mouse Events	273
Keyboard Events	274
Window Events	274
Switch Events	275
User-Defined Events	275
Desk Accessory Events	275
The Null Event	275
Event Priorities	276
Event Records	277
What — Event Codes	277
Message — Event Message	279
When — Elapsed Time	279
Where — Mouse Pointer Location	279
Modifiers — Modifier Flags	279
Event Manager Tool Calls	281
Housekeeping Calls	282
Calls that Access Events	283
Mouse-Reading Calls	286
A Simple Program that Uses the Event Manager	286
What's Next?	293
Chapter 9 — Working with Windows	294
Window Components	295
Content Region	295
Title Bar	295
Close and Zoom Boxes	295
Grow Box	296
Scroll Bars	297
Information Bar	298
Active and Inactive Windows	299
Fundamental Tool Calls for Windows	300
NewWindow	300
ShowWindow	306
CloseWindow	307
Window-Shuffling Calls	307
The TaskMaster	307

Calling TaskMaster	308
Processing Events	309
Tool Sets Required by TaskMaster	311
An Example Window Program	311
Chapter 10 — Menus	326
Menu Bars and Pull-Down Menus	326
The System Menu Bar	327
Pull-Down Menus	328
Enabled and Disabled Menus	328
Menu Items	329
Creating Menu Bars and Menus	331
The Menu/Item Line List	332
Menu Modifiers	334
Responding to Menu Events	337
Mouse Events	337
Responding to Mouse Events	338
Key Events	338
Providing for New Desk Accessories	340
An Example Program that Provides Menus	340
Chapter 11 — Controls	356
Predefined Controls	357
Buttons	357
Check Boxes	358
Radio Buttons	358
Scroll Bars	358
Scroll Bar Components	358
Active and Inactive Controls	359
Control Manager Tool Calls	360
Chapter 12 — Conducting Dialogs	362
Dialog Boxes	363
Modal Dialog Boxes	363
Modeless Dialog Boxes	363
Alert Boxes	364
Types of Alert Boxes	365
Stages of an Alert	366
Programming Dialogs and Alerts	366
Item Lists	367

ID Number	368
Display Rectangle	368
Item Type	368
Item Descriptor and Value	371
Item Flag	372
Tool Calls for Dialog Boxes	372
Handling Modal Events	375
Handling Modeless Events	376
Get Text	376
Example Dialog Box Program	378
Tool Calls for Alert Boxes	393
Alert Template	393
Item Template	395
Final Comments	396
Appendix A — Hexadecimal/Decimal Conversion	397
Appendix B — ASCII Table	398
Appendix C — 65816 Instruction Set Summary	399
Appendix D — Requirements for Using Tool Sets	432
Loading RAM-Based Tools from Disk	432
Working Space for Tool Sets	434

Preface

Why Assembly Language?

Many people write all of their computer programs in one of the so-called high-level languages, particularly BASIC. BASIC is easy to learn, easy to use, and fast enough for most computing tasks. That being the case, why would anyone want to use any other language? One reason is that BASIC, like human languages, is not well-suited to everything. Some tasks are much easier in other languages. Imagine, for example, trying to describe fine cooking without some French words, or symphonies without some Italian terms. Similarly, special computing tasks like graphics, music, or word processing are often easier in other languages.

Furthermore, BASIC is quite slow. The term slow may surprise the beginner, since most programs seem to run instantaneously. However, BASIC tends to fall short in the following situations:

1. When large amounts of data are involved. Notice how slow BASIC is when a program must, for example, sort a long list of names and addresses or accounts. Similarly, BASIC is quite slow when a program must search through a 50-page report or keep inventory records on thousands of items.
2. When graphics are involved. If a program is drawing a picture on the screen, it must work quickly or the delay is intolerable. If objects in the picture are supposed to move, the program must be fast enough to make the motion look natural. This is particularly

difficult when the picture contains many objects (such as spaceships, base stations, and alien invaders), all of which are moving in different directions.

3. When a lot of decisions or “thinking” is required. This is often necessary in complex games like checkers or chess. The program has to try many possibilities and decide on a reasonable move. Obviously, the more possibilities there are and the more analysis required, the longer it will take the computer to move.

Why is BASIC slow? In the first place, the computer actually translates each BASIC statement into one or more simple internal commands (so-called machine language). It does this every time it runs the BASIC program. Thus, much of the computer’s time is spent decoding the program, not running it.

There are versions of BASIC called *compilers* that perform the translation once and then save the translated version. However, unless the compiler is efficient at “optimizing” programs, even a compiled BASIC program may be slow. A BASIC program is really like an automobile with an automatic transmission; no amount of coaxing can get you the performance or fuel economy that a skillful driver can achieve with a manual transmission. The human being is simply a more flexible, more skillful, and smarter operator than the automatic transmission or the BASIC interpreter or compiler.

Assembly language is like a manual transmission. It gives the programmer greater control over the computer at the cost of more work, more detail, and less convenience. Like an automatic transmission, BASIC is good enough for most programmers. But for those who must get optimum performance from their computers, assembly language is essential. You will find that most complex games, graphics programs, and large business programs are written at least partially in assembly language.

Even if assembly language is your likely choice, you may be wondering if you have enough background to learn assembly language programming. You *do* if you have done some programming of *any* kind. If you know BASIC, Pascal, C, or some other so-called “high-level” language, that’s even better.

The Contents of This Book

For the benefit of former users of high-level languages, this book has two starting points. If you have never programmed in assembly language,

read Chapter 0, which gives a “crash course” in the *binary* and *hexadecimal* numbering systems. Otherwise, if you already know what these terms mean and understand how to use them, proceed directly to Chapter 1.

Chapter 1 describes the 65816 microprocessor — the “brain” of the Apple IIGS — and the major hardware components inside the IIGS. It also provides a general introduction to assembly language programming and gives an overview of programming the Apple IIGS using its *Toolbox*.

Chapter 2 discusses assemblers in general and then describes the *Apple IIGS Programmer's Workshop*, a software development package offered by Apple Computer. Chapter 2 also presents a simple program and tells how to enter it into the computer, assemble it, and execute (run) it. It also describes the Workshop's *Debugger*, a utility that helps you track down errors in your programs.

Chapter 3 describes the 65816 *addressing modes* you can use to access the data on which your program is to operate.

Chapter 4 discusses the 65816's instruction set, the assembly language commands you can use to communicate with the IIGS. This book treats the instructions in functional groups, rather than alphabetically. That is, I have grouped add with subtract, load with store (the assembly language equivalents of BASIC's Peek and Poke, respectively), and so on. Through this approach, you not only get to *understand* what the instructions do, but you also appreciate how they fit together.

Chapter 5 covers *macros*. A macro is a miniprogram that you can insert in a main program simply by mentioning its name. Macros can make assembly language programs nearly as easy to develop as BASIC programs, and the Programmer's Workshop disk contains *hundreds* of them.

Chapter 6 surveys the Apple IIGS *Toolbox* and the *tool sets* within it. There are tool sets that produce graphics, sound, windows, menus, and virtually any other feature you might want to include in a program. Chapter 6 also includes a *program model* that contains the “boilerplate” programs needed to communicate with the IIGS.

Chapter 7 shows you how to display graphics objects on the screen using the built-in *QuickDraw II* tool set. With QuickDraw, you can easily produce predefined shapes such as rectangles and circles, or objects of your own design.

Chapter 8 tells how to deal with “events,” such as the user pressing a key or the mouse button.

Chapter 9 discusses on-screen *windows* and the “controls” the user can employ to operate on them. Chapter 10 covers *menus* that appear on the screen to let the user select a course of action.

The final two chapters are also concerned with user interactions. Chapter 11 provides a brief introduction to buttons, check boxes, and other controls provided by the Control Manager. Chapter 12 shows how to use those controls along with other on-screen items to conduct a conversation or *dialog* with the user.

The book provides four appendixes for your convenience. Appendix A has tables that help you convert hexadecimal numbers to decimal, and vice versa. Appendix B shows the text characters you can display on the screen. Appendix C summarizes the 65816's instruction set in alphabetical order, and shows the legal form for each instruction, how long it takes to execute, how many bytes it occupies in memory, and which status flag it affects. Finally, Appendix D tells what's required to use the tool sets; that is, how they interact and how much working space in memory they need.

For Further Reference

The Apple IIGS is a powerful computer that can do virtually any programming task you want, and I have attempted to describe its most important programming features in the order people generally use them. However, as you gain experience, you will probably want to know more about this fascinating machine. The full details are available in a comprehensive set of technical manuals. They are:

Technical Introduction to the Apple IIGS
Programmer's Introduction to the Apple IIGS
Apple IIGS Hardware Reference
Apple IIGS Firmware Reference
Apple IIGS Toolbox Reference (Volumes 1 and 2)
Apple IIGS Programmer's Workshop
Apple IIGS Programmer's Workshop Assembler Reference
Apple IIGS ProDOS 16 Reference
Apple IIGS Human Interface Guidelines

These manuals are available from your Apple dealer or from the Apple Programmer's and Developer's Association (APDA): 290 SW 43rd Street, Renton, WA 98055.

At the very least, you should have the *Technical Introduction*, the *Programmer's Introduction* and both volumes of the *Toolbox Reference*, to help you proceed from where this book leaves off.

I also recommend two other books in Bantam's "Apple IIGS Library" series. The first, *The Apple IIGS Book* by Jeanne DuPrau and Molly Tyson, is an excellent general-purpose reference on all aspects of the IIGS. Written by two Apple insiders, the book also contains the complete history of the IIGS, including some interesting (and often humorous) anecdotes about the people who designed it. For example, the authors reveal that "GS" stands for . . . are you ready for this? . . . Gumby Software(!), after The Man of Clay, the official mascot of the IIGS design team.

Danny Goodman's *The Apple IIGS Toolbox Revealed* is another worthwhile addition to your library. Danny's plain-English treatment of the Toolbox and the tool sets within it is first-rate. I refer to it often and I'm sure you will, too.

CHAPTER 0

About Computer Numbering

Unless you're visiting from another planet (and if so, welcome!), you have spent your entire life counting things using decimal numbers. Mathematicians call decimal the *base 10* numbering system because it has ten digits, 0 through 9.

Humans are comfortable counting in decimal (probably because we have ten fingers and ten toes), but computers are not. Instead, they count with the base 2, or *binary*, numbering system. The binary system has only two digits, 0 and 1. Hence, to communicate with a computer at its own level — as you do when you program in assembly language — you must be familiar with binary numbering. Besides binary, assembly language programmers also use the base 16, or *hexadecimal*, numbering system, so you must be familiar with it as well. The hexadecimal system has 16 digits, 0 through 9 and A through F.

This chapter is a “crash course” in computer numbering systems, for readers who have had no previous exposure to them. That's why I numbered it Chapter 0. If you already understand binary and hexadecimal numbering, feel free to skip directly to Chapter 1.

Binary Numbering

A computer gets all program instructions and data from its memory. Memory consists of integrated circuits (or “chips”) that contain thousands of electrical components. Like light switches and the power switch on your computer, these components have only two possible settings: “on” and “off.” Still, with only these two settings, combinations of memory components can represent numbers of any size.

The on and off settings of a memory component correspond to the 1 and 0 digits of the base 2 or binary numbering system. The switch-like components of memory are called “bits,” short for *binary* digits. By convention, a bit that is on has the value 1, while a bit that is off has the value 0. This appears woefully limiting until you consider that a decimal digit (no, it’s not called a “det”) can only range from 0 to 9. Just as one can combine decimal digits to form numbers larger than 9, one can also combine binary digits to form numbers larger than 1.

As you know, to represent a decimal number larger than 9, you must attach an additional “tens position” digit; to represent a number larger than 99, you must attach a “hundreds position” digit, and so on. Each decimal digit you add “weighs” 10 times as much as the digit to its immediate right.

For example, 324 can be represented as

$$(3 \times 100) + (2 \times 10) + (4 \times 1)$$

or as

$$(3 \times 10^2) + (2 \times 10^1) + (4 \times 10^0)$$

Thus, reading right to left, each decimal digit is a power of 10 greater than the preceding digit.

The binary numbering system works the same way, except *each binary digit is a power of 2 greater than the preceding digit*. That is, the rightmost bit has a weight of 2^0 (decimal 1), the second bit has a weight of 2^1 (decimal 2), the third bit has a weight of 2^2 (decimal 4), and so on. For example, the binary value 1001 has a value of decimal 9 because:

$$1001_2 = (1 \times 2^3) + (1 \times 2^0) = (1 \times 8) + (1 \times 1) = 9^{10}$$

In short, to find the value of any given bit position, double the weight of the preceding bit position. Thus, the weights of the first 8 bits are 1, 2, 4, 8, 16, 32, 64, and 128, as shown in Figure 0-1.

7	6	5	4	3	2	1	0	Bit position
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	Power of 2
128	64	32	16	8	4	2	1	Decimal value

Figure 0-1

Converting Decimal Values to Binary

To convert a decimal value to binary, you make a series of simple subtractions, where each subtraction produces the value of a single binary digit (bit). To begin, subtract the largest possible power of 2 from the decimal value and enter a 1 in the corresponding bit position. Then subtract the next largest possible power of 2 from the result and enter a 1 in *that* bit position. Continue making subtractions until the result is zero. Enter a 0 in any bit position whose weight cannot be subtracted from the current decimal value.

For example, to convert decimal 52 to binary:

$$\begin{array}{rcl}
 52 & & \\
 \underline{-32} & \text{bit position 5} = 1 & \\
 20 & & \\
 \underline{-16} & \text{bit position 4} = 1 & \\
 4 & & \\
 \underline{-4} & \text{bit position 2} = 1 & \\
 0 & &
 \end{array}$$

Entering a 0 in the other bit positions (3, 1, and 0) yields the final binary result of 110100.

To verify that decimal 52 is indeed binary 110100, add the decimal weights of the “1” positions:

$$\begin{array}{rcl}
 32 & (\text{bit 5}) & \\
 16 & (\text{bit 4}) & \\
 + 4 & (\text{bit 2}) & \\
 \hline
 52 & &
 \end{array}$$

4 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

Bytes

The Apple II series, Commodore 64, Tandy/Radio Shack TRS 80, and many other popular microcomputers are designed around *8-bit* microprocessors. Eight-bit microprocessors are so named because they transfer data eight bits at a time. To transfer more than eight bits, they must perform additional operations.

In computer terminology, an 8-bit unit of information is called a *byte*. With eight bits, a byte can represent decimal values from 0 (binary 00000000) to 255 (binary 11111111).

Because bytes are a convenient unit of data, microcomputers are usually described in terms of the number of bytes (rather than bits) their memories can hold. Manufacturers typically organize memory in blocks of 1,024 bytes. This particular value reflects the binary orientation of computers in that it represents 2^{10} bytes.

The value 1,024 has a standard abbreviation: the letter *K*. Hence a computer that has a “512K memory” contains 512×1024 (or 524,288) bytes.

Adding Binary Numbers

You can add binary numbers the same way you add decimal numbers: by “carrying” any excess from one column to the next. For example, adding the decimal digits 7 and 9 produces 6 in the “ones” column and an excess 1 that you must carry into the “tens” column. Similarly, adding the binary digits 1 and 1 produces a 0 in the “ones” column and an excess 1 that you must carry to the “twos” column.

Adding multibit binary numbers can be somewhat more complex, because you must keep track of multiple carries. For example, this operation involves two carries:

$$\begin{array}{r} 1011 \\ + 11 \\ \hline 1110 \end{array}$$

Adding the rightmost column ($1 + 1$) produces a result of 0 and a carry of 1 into the second column. With the carry, adding the second column ($1 + 1 + 1$) produces a result of 1 and a carry of 1 into the third column. Table 0-1 summarizes the general rules for binary addition.

Table 0-1

Inputs			Results	
Operand #1	Operand #2	Carry	Sum	Carry
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	0	1	0	1
1	0	1	0	1
1	1	1	1	1

Signed Numbers

Until now, I have concentrated on unsigned binary numbers, in which each bit has a weight that reflects its position. However, sometimes you want to operate on positive or negative values; that is, on *signed* numbers.

When a byte contains a signed number, only the seven low-order bits (0 through 6) represent data; the high-order bit (7) specifies the sign of the number. *The sign bit is 0 if the number is positive or zero, and 1 if it is negative.* Figure 0-2 shows the arrangement of unsigned and signed bytes.

A byte that holds a signed number can represent positive values between 0 (binary 00000000) and +127 (01111111) and negative values between -1 (11111111) and -128 (10000000). Clearly, the way negative numbers are represented needs an explanation.

Note that -1 in binary is 11111111. Wouldn't it be simpler to make it just 10000001 (that is, 1 with a minus sign bit)? It might be simpler, but it would also produce wrong answers. For example, consider what would happen if you were to add +1 and -1. The answer should, of course, be 0, but instead you would get:

$$\begin{array}{rcl}
 00000001 & = & +1 \\
 10000001 & = & -1 \\
 \hline
 10000010 & = & +2 (!)
 \end{array}$$

Thus, what we need is some way to represent -1 so that adding +1 and -1 produces 0. Indeed, that's how mathematicians arrived at 11111111 to represent -1: it produced the right answer. Now the preceding addition becomes:

6 APPLE II GS ASSEMBLY LANGUAGE PROGRAMMING

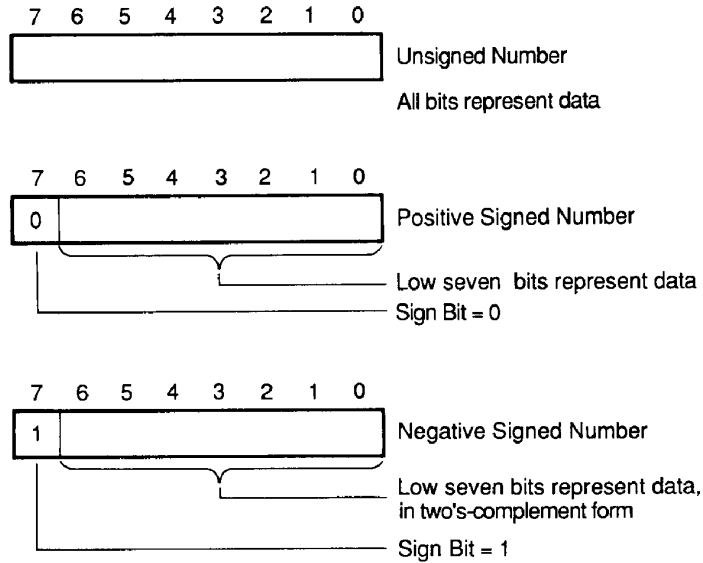


Figure 0-2

$$\begin{array}{rcl} 00000001 & = & +1 \\ 11111111 & = & -1 \\ \hline 1\ 00000000 & = & -0 \end{array}$$

The extra 1 bit at the front is a carry that's been left over from the addition. Since we are operating on 8-bit numbers, we simply ignore this ninth bit.

Two's-Complement

Like -1 , all negative signed numbers are represented in a special form that makes additions and subtractions produce the right answers. This is called the *two's-complement* form.

To find the binary representation of a negative number (that is, to find its two's-complement), simply take the positive form of the number and reverse each bit — change each 1 to a 0 and each 0 to a 1 — then add 1 to the result. The following example shows how to calculate the binary representation of -32 in two's-complement form:

00100000	+32
11011111	Reverse every bit
+ 1	Add 1
11100000	+32 in two's complement form

Of course, the two's-complement convention makes negative numbers difficult to decipher. Fortunately, you can use the procedure I just gave to find the positive form of a (two's-complemented) negative number. For example, to find what value 11010000 has, proceed as follows:

00101111	Reverse every bit
+ 1	Add 1
00110000	= 16 + 32 = +48

You'll be happy to hear that you don't normally have to conduct this kind of exercise too often, because the assembler lets you enter numbers in decimal form (signed or unsigned) and does all the converting automatically. However, sometimes you may want to interpret a negative number that is stored in a register or memory, so you should know how to make these conversions manually.

Hexadecimal Numbering

Although computers process only binary values, working with strings of nothing but ones and zeroes can be maddening for humans. It's also easy to induce errors with them, because it's extremely easy to mistype something like 10110101. A single misplaced 1 or 0 can ruin all your calculations.

Years ago, programmers found that they were generally operating on *groups* of bits rather than individual bits. The first microprocessors were 4-bit devices (they processed data four bits at a time), so the reasonable alternative to binary was a system that numbered bits in groups of four.

As you now know, four bits can represent the binary values 0000 through 1111 — decimal 0 through 15 — a total of 16 possible combinations. If a numbering system is to represent 16 combinations, it must have 16 different digits. That is, it must be a *base 16* numbering system.

If base 2 numbers are called "binary" and base 10 numbers are called "decimal," what term is appropriate for the base 16 system? Well, whoever

Table 0-2

Hexadecimal Digit	Binary Value	Decimal Value	Hexadecimal Digit	Binary Value	Decimal Value
0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	A	1010	10
3	0011	3	B	1011	11
4	0100	4	C	1100	12
5	0101	5	D	1101	13
6	0110	6	E	1110	14
7	0111	7	F	1111	15

named the base 16 system combined the Greek word *hex* (for six) with the Latin word *decem* (for ten) to form the word *hexadecimal*. Hence the base 16 system is called the hexadecimal numbering system.

Of the 16 digits in the hexadecimal numbering system, the first ten are labeled 0 through 9 (as in the decimal system), while the last six are labeled A through F (decimal 10 through 15). Table 0-2 lists the binary and decimal values for each hexadecimal digit.

Like binary and decimal digits, each hexadecimal digit has a weight that is some multiple of its base. Since the hexadecimal system is based on 16, each digit weighs 16 times more than the digit to its immediate right. That is, the rightmost digit has a weight of 16^0 , the next has a weight of 16^1 , and so on. For example, the hexadecimal value 3AF has the decimal value 943 because

$$(3 \times 16^2) + (A \times 16^1) + (F \times 16^0) \\ (3 \times 256) + (10 \times 16) + (15 \times 1) = 943$$

While BASIC and other high-level languages usually display numbers in decimal form, the utilities that programmers use to develop assembly language programs generally display numbers in hexadecimal form. This includes memory addresses, instruction codes, and data. Therefore, to get maximum benefit from your programming, try to “think hexadecimal.” This is difficult at first, but it will become easier as you gain experience. To help you along, Appendix A provides a table for converting between decimal and hexadecimal.

CHAPTER 1

Introduction to Assembly Language

What is Assembly Language?

Like BASIC, assembly language is a set of commands that tell the computer what to do. However, the commands in the assembly language instruction set refer to computer components directly. It's like the difference between telling someone to walk down to the corner and telling them precisely how to move their muscles and maneuver past obstacles. Obviously, a simple command is sufficient most of the time; only athletes and mountain climbers need the more detailed instructions.

Assembly language programs give the computer detailed commands, such as "load 32 into the X register," "copy the contents of the A register into the Y register," and "store the number in the Y register into memory location 300." As you see, BASIC and assembly language differ in how you instruct the computer. With BASIC, you speak in *generalities*; with assembly language, you speak in *specifics*.

Although assembly language programs take more time and effort to write than BASIC programs, they also run much faster. The level of detail

is the key here. The idea is the same as an athlete who runs faster or jumps further by watching every step of what he or she does. Precise form is essential to achieving maximum performance.

Because assembly language requires you to make direct use of the computer's internal components, you must understand the features and capabilities of the integrated circuit (or "chip") that contains these components, the computer's *microprocessor*. The microprocessor inside the Apple IIGS is a *Western Design Center 65C816*. (The "C" stands for CMOS, short for Complimentary Metal-Oxide Semiconductor. CMOS is the process used to manufacture the chip.) In this book, I'll call it the 65816 — or sometimes just 816 — simply because it's easier to read.

Overview of the 65816 Microprocessor

The 65816 is a 16-bit version of the 8-bit 6502, the microprocessor inside the Apple II series and many other personal computers. However, I don't mean to imply that size is the only difference between the two microprocessors. That would be like saying a Cadillac is an 8-cylinder version of a Volkswagen. The 65816 has quite a few powerful features that the 6502 does not have, and I will note them when necessary.

Data and Address Buses

The 65816 always transfers information to or from memory and I/O devices 8 bits at a time, on its 8-line *data bus*. (Normally, a processor that has an 8-bit data bus would be classified as an 8-bit device. The 65816 is called a 16-bit microprocessor because all its internal circuitry is 16 bits wide.)

Note that I say "memory *and* I/O devices". Like the 6502, the 65816 does not distinguish between memory and I/O — it treats all external devices the same. Hence, it has no special input or output instructions; it has only "load" and "store" instructions, and you use them to transfer data to any external device, regardless of whether it is memory or a peripheral. Like each memory location, each peripheral device has a unique address; the address determines where the 816 sends the data or obtains it.

The 65816 transmits addresses to external devices over 24 lines that are collectively called the *address bus*. Being 24 bits wide, the address bus allows the 816 to access up to 16,777,216 bytes, or *16 megabytes* (abbreviated 16M) — the same range as an IBM System/370! Contrast this with

the 6502, which can address only 64K bytes directly. (Old-timers may recall when 64K was considered a lot of memory!)

Memory Organization

Because the 65816 must be able to work with so much more memory than the 6502 (16M bytes versus 64K bytes), accessing the 65816's memory is slightly more complex.

In the 6502, memory is divided into 256-byte *pages*. Since the 6502 can address up to 64K directly, there are 256 pages in all. Page 0 occupies addresses 0 to 255, page 1 occupies addresses 256 to 511, and so on. (The 6502 treats the lowest page in memory — page 0 or the *zero page* — differently from other pages. The 65816 has an equivalent, special purpose page, called the *direct page*, in low memory. I'll discuss it later.)

If you have a recent Apple //e, you may be thinking, "Scanlon doesn't know what he's talking about. My //e has a 6502, yet it has a 128K memory." Ah, but reread the preceding paragraph, where I state "... the 6502 can address up to 64K bytes *directly*". Apple's engineers gave the //e the ability to access 128K bytes by installing circuits that switch between two 64K *banks* of memory. Bank 0 is the lower 64K locations, while bank 1 is the upper 64K. Only one bank can be active at any given time, so my statement about the 6502 addressing only 64K still stands.

The 65816 uses the same page-and-bank scheme as the 6502, except it has bank-switching circuitry built in. Specifically, it has two bank selection registers, where one selects the bank that contains program instructions and the other selects the bank that contains data. With its 16-megabyte addressing range, the 816 can address up to 256 banks. (It's no coincidence that both pages and banks involve the number 256. As I mentioned in Chapter 0, the value 256 and other multiples of 2 reflect the binary nature of computers.) Just because the computer *can* address this much memory doesn't mean that it's all actually available. For example, the standard 256K model of the Apple IIGS has only four banks of read/write memory.

Software Features

The 65816 provides 91 types of assembly language instructions, 35 more than the 6502. It also provides 24 addressing modes, 11 more than the 6502. (An addressing mode is a technique the microprocessor uses to obtain a number it is to add, subtract, or whatever.)

How fast does an instruction execute? That depends on which instruction you're referring to, which addressing mode you're using, and how fast

the microprocessor is operating. Like electronic watches, microprocessors are regulated by quartz crystals. The crystal emits pulses at a fixed rate, which determines how fast the microprocessor operates. In the Apple IIGS, the crystal emits either 1.0 million or 2.5 million pulses per second, depending on which operating speed is active in the Control Panel.

Computerists don't refer to pulses per second, however, but to *cycles per second* or, more often, *Hertz*. Pulses per second, cycles per second, and Hertz all mean the same thing, but in this book I use the term Hertz or its abbreviation, Hz. Hence, one can say that the clock in the IIGS normally runs at 2.5 million hertz, or 2.5 MHz (short for megahertz). At 2.5 MHz, the 816's clock "ticks" every 400 *nanoseconds*, where a nanosecond (abbreviated ns) is one-billionth of a second, or 10^{-9} seconds. At 1.0 MHz, the clock "ticks" every 1,000 nanoseconds; 1,000 nanoseconds, which is one millionth of a second or 10^{-6} seconds, is called one *microsecond* (μ s).

The fastest instructions — for example, those that increment a register — execute in 2 cycles, or 800 ns at 2.5 MHz. The slowest instructions — for example, ones that call a subroutine — take 8 cycles, or 3,200 ns at 2.5 MHz. (Note that even the "slow" instructions execute in the remarkable time of 0.0000032 seconds!) Most instructions require between 3 and 6 cycles to execute.

Interrupts

The microprocessor in a computer (an Apple or any other) does not simply run programs. As the chief regulator of the system, it gets involved in one way or another with everything that happens. For instance, when someone presses a key at the keyboard, the processor must find out which key was pressed and do whatever is appropriate for that particular key. Similarly, when a disk drive is transferring data to or from the computer's memory, the processor is responsible for carrying out the instructions that make that transfer happen. As I just said, the microprocessor has a role in *everything* the computer does.

Well, how does a microprocessor get involved with a peripheral device? It doesn't have ESP, so it can't *know* which device needs attention. Then again, it shouldn't sit there asking or "polling" each peripheral whether it needs something. If the processor polled peripherals all day, it wouldn't be able to do anything else — it would have no time to run programs. This would be like having a telephone with no bell. You would have to pick up the receiver every so often just to find out whether anyone was on the line!

In fact, microprocessors and peripherals communicate in a very efficient

fashion. The microprocessor continues to run a program (say, DOS or BASIC or a word processor) until the keyboard, display unit, or some other peripheral says, "Excuse me, micro, but I need your help in getting something done. Would you please stop what you're doing long enough to help?" Of course, peripherals don't actually talk to the microprocessor; they send a special "help me" signal called an *interrupt request*.

Interrupts operate differently between various microprocessors, but in the 65816 (and 6502 and 65C02), here is what happens. When a peripheral device needs servicing, it activates an interrupt request line that is connected to the 816. There is only one interrupt request line; all devices in the system share it.

So, in essence, here is a device ringing the processor's doorbell. Sometimes the processor is doing something so important that it can't stop to respond to the request. In that case, the 816 simply says, "Let the bell ring. I'll answer when I can."

Otherwise, if the processor is doing a job that can wait until later, it makes some notes about the job (so it knows where to resume later), then reads a new program address from a special *interrupt vector* in memory and starts executing that program. The program determines which device is making the request, then transfers to the routine for that particular device. When the 816 finishes servicing the peripheral, it uses the notes it made earlier to get back to the original job.

Operating Modes

The 65816 is actually two microprocessors in one, insofar as it can operate in two different modes, called *native* and *emulation*. In the normal operating mode, native, the 816 can process data either 8 or 16 bits at a time. In emulation mode, it acts like a 6502, and can only process 8-bit data. When you switch the Apple IIGS on, the 65816 starts in native mode; it stays in this mode until a program explicitly switches it to emulation mode.

Internal Registers

Figure 1-1 shows the 65816's internal registers and, because it can operate in emulation mode, the 6502's registers. Note that the 6502's data registers (A, X, and Y) and status register (P) are 8 bits long, whereas its address registers (S and PC) are 16 bits long. This is typical for an 8-bit microprocessor; it has an 8-bit data bus and a 16-bit address bus. (Incidentally, the

14 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

shaded "01" that represents the high-order byte of the stack pointer is a hexadecimal value. It indicates that the 6502's stack is always located in page 1 of memory. More about stacks later.)

As you can see, the 65816 microprocessor provides all the 6502 data registers (as the low bytes of its own data registers) and address registers (as the low 16-bit words of its own address registers), but extends data registers to 16 bits and address registers to 24 bits.

Lest you be mistaken, I must emphasize that the 65816 contains only *one* set of registers: those shown in the right-hand column of Figure 1-1. When emulating a 6502, however, the 816 uses those registers as a 6502 would. It interprets any program reference to the A or X register as meaning AL or XL, limits the stack pointer to 16 bits (and puts hex 01 in the upper

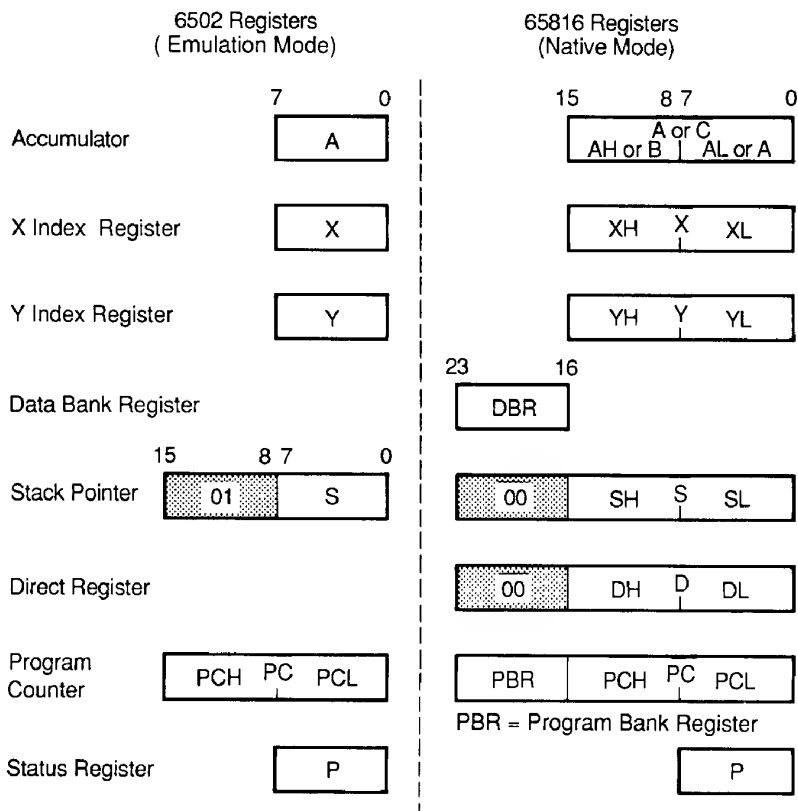


Figure 1-1

byte), and so on. In emulation mode, then, the 816 acts like an adult speaking to a child. It uses only the vocabulary the child can understand, which is something less than the full vocabulary the adult knows.

The following sections describe only the 65816's regular registers, the ones it uses when operating in native mode. To learn about the 6502 registers, which are active in emulation mode, buy one of the many 6502 assembly language books on the market.

General-Purpose Registers

You can treat these registers as either three 16-bit registers or six 8-bit registers, depending on whether you're operating on 16-bit words or 8-bit bytes. The 16-bit registers are named A, X, and Y. The 8-bit registers within them are named AH, AL, XH, XL, YH, and YL. In each case, "H" and "L" indicate the high-order and low-order bytes of the 16-bit registers.

A, the accumulator, is the main register for performing arithmetic and logical operations; it holds one operand and the result of most of these operations. *X* and *Y* serve primarily as index registers to calculate a memory address. However, the 816 also provides instructions that increment and decrement *X* and *Y*, which make them useful as counters.

Data Bank Register

The 8-bit data bank register (DBR) provides the bank number in addressing modes that otherwise generate only the lower 16 bits of an address. There, DBR acts as an 8-bit "extension" of the *X* or *Y* register that effectively makes *X* or *Y* 24 bits long. When you switch the Apple II GS on, the 65816 initializes DBR to zero, thereby selecting bank 0.

Stack Pointer

As with most other microprocessors, the 65816's memory contains a special data structure called a *stack* that serves as a temporary holding area for data and addresses. When you execute, or "call," a subroutine, the 816 uses the stack to hold the return address — a place marker that eventually brings the processor back to its original spot in the program. You can also use the stack to preserve the contents of registers that a subroutine alters.

The 65816 enters data onto the stack and extracts data from it the same way you (or, with any luck, your children) stack dishes in your kitchen; that is, the last item to be placed on the stack is also the first item to be removed from it. Computerists usually refer to this type of stack as "last in, first out"

(or LIFO). As data items are *pushed* onto the stack, they are stored in memory at ever-lower addresses; the stack “builds” toward address 0.

The stack pointer (S) is a 16-bit register that points to the next available location on the stack; it is the stack’s maitre d’. The 65816 decrements S by 1 as each new byte is pushed onto the stack and increments it by 1 whenever a byte is *pulled* off the stack.

As the shaded “00” in Figure 1-1 indicates, the stack is always in bank 0. Note, however, that while the 6502’s stack must be in page 1, the 65816’s stack can be in any page. In general, though, you shouldn’t worry about where the stack is located, what the stack pointer contains, or what’s on the stack (unless you specifically stored something there). The computer’s software handles all stack operations, so unless you do something awful — such as push two items on the stack but extract only one — you should have no problems.

Direct Register

Recall under “Memory Organization” that I mentioned a special-purpose *direct page* in bank 0. The 24-bit direct register (D) determines which block of 256 bytes is to be used as the direct page. When you switch the Apple IIGS on, the 65816 initializes the direct register to 0.

Program Counter

The 65816 executes (runs) programs by obtaining instructions from memory, one at a time. The *program counter* (PC) determines which memory location the 816 will access next. The 816 increments the PC automatically after each memory access so that it points to the next consecutive location. Because the program counter is 16 bits wide, it can access any location in the active 64K bank. A separate program bank register (PBR) specifies the bank.

Like the stack pointer, the program counter is a register that the microprocessor uses to keep track of addresses (in this case, the addresses of instructions), and you shouldn’t be concerned with it unless you’re troubleshooting or “debugging” a program.

Program Bank Register

The 8-bit program bank register (PBR) provides the bank number for the instruction that the 816 is to execute next. Thus, it “extends” the program counter to 24 bits. When you switch the Apple IIGS on, the 65816 initializes PBR to zero, thereby selecting bank 0.

Processor Status Register

The 8-bit processor status (P) register contains one-bit indicators. Some of these bits are *status flags* that reflect the result of a previous instruction (generally, the preceding instruction), others are *mode control bits* that determine how the 65816 operates. The P register can take either of two forms, depending on whether the 816 is operating in emulation mode (Figure 1-2) or native mode (Figure 1-3).

In *native mode*, the P register bits do the following:

Bit 0 — The *carry (C) flag* is 1 if an addition produces a carry or a subtraction produces a borrow; otherwise, C is 0. C also holds

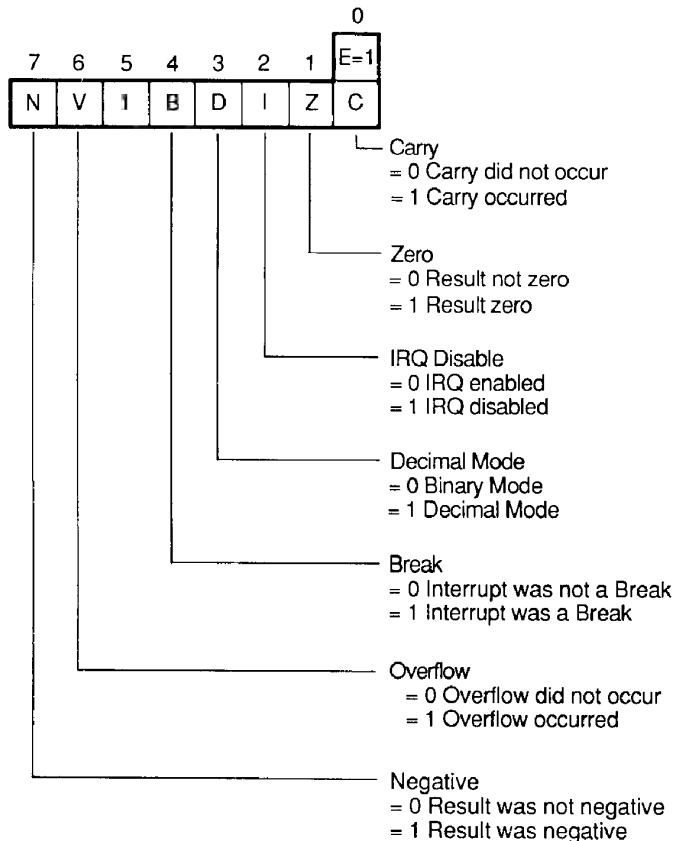


Figure 1-2

18 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

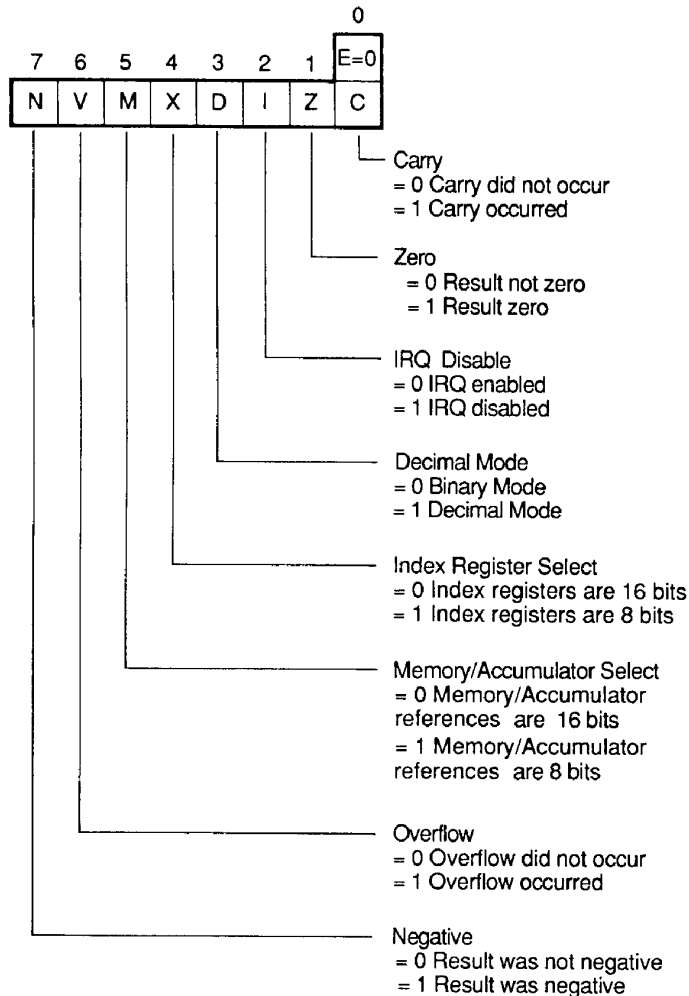


Figure 1-3

the value of a bit that has been shifted or rotated out of a register or memory location, and reflects the result of a compare operation.

Bit 1 — The *zero (Z) flag* is 1 if the result of an operation is zero; a nonzero result clears Z to 0.

- Bit 2** — The *IRQ disable (I) flag* allows the 65816 to recognize interrupts from external devices in the system. Setting I to 1 makes the 816 ignore interrupt requests until I becomes 0.
- Bit 3** — The *decimal mode (D) flag* controls whether the 65816 is operating on binary numbers (0) or decimal numbers (1). In the decimal mode, the 816 treats arithmetic operands as binary-coded decimal (BCD) digits “packed” two per byte.
- Bit 4** — The *index register select bit (X)* specifies whether X and Y are to be treated as 16-bit registers (0) or 8-bit registers (1). Switching the X bit from 0 to 1, or vice versa, leaves XL and YL unchanged, but clears XH and YH to 0.
- Bit 5** — The *memory/accumulator select bit (M)* specifies whether operands in memory or the accumulator are 16-bit values (0) or 8-bit values (1). Switching M from 0 to 1, or vice versa, has no effect on AH (B) or AL (A).
- Bit 6** — The *overflow (V) flag* is an error indicator for operations on signed numbers. V is 1 if adding two like-signed numbers or subtracting two opposite-signed numbers produces a result that the operand can’t hold; otherwise, V is 0.
- Bit 7** — The *negative (N) flag* is meaningful only for operations on signed numbers. N is 1 if an arithmetic, logical, shift, or rotate operation produces a negative result; otherwise, N is 0. In other words, N reflects the most-significant bit of the result, regardless of whether it is 8 or 16 bits long.

Unless you are an experienced programmer or a daring one, I suggest you always leave the M and X bits set to 0. Setting either or both to 1 sometimes produces unexpected results.

The processor status bits are the same in *emulation mode*, except bit 4 is a break command flag and bit 5 is unused; it’s always 1. The break command (B) flag indicates whether an interrupt request to the processor was generated by a BRK instruction (1) or by an externally-generated interrupt (0).

Inside the Apple II GS

Although you must understand the internal architecture of the 65816 to be an effective assembly language programmer, you needn’t know much about the

hardware within the Apple IIGS. Still, a general understanding of the hardware can't hurt. In fact, you will probably feel more comfortable programming the IIGS if you know what goes on inside it.

Figure 1-4 shows an engineering-style block diagram of the major hardware components in the Apple IIGS. Note that the diagram is divided into a "slow" and "fast" side. On the slow side are components that can only run at 1.0 MHz, the standard Apple II operating speed. The fast side has the components that can run at 2.5 MHz, the speed at which the IIGS operates in native mode.

Fast (2.5-MHz) Components

The "fast" side of the block diagram includes the 65816 microprocessor, a fast processor interface (FPI) chip, a 128K block of read-only memory (ROM), two 64K banks of RAM, and a connector for expanding RAM beyond its standard 256K bytes.

The FPI chip regulates the computer's operating speed. It sends out a 2.5-MHz clock signal when the IIGS is running in native mode and a 1-MHz clock when the IIGS is running in emulation mode. The FPI also synchronizes the processor running at 2.5 MHz with circuits on the "slow" side running at 1 MHz.

The 128K ROM holds all of the computer's built-in programs, or *firmware*. ROM is non-volatile; its contents stay intact even when the power is off. ROM includes, of course, the program that puts the IIGS into some predefined initial state (native mode, registers set to zero, and so on) when you switch it on. But ROM also includes many more "goodies" that I describe later.

The two banks of RAM are, in fact, the banks that regular Apple II programs use. Since Apple IIs are 1-MHz computers, why is their memory on the fast (2.5-MHz) side? The answer to that question is somewhat involved, so I'll postpone discussing it until later.

Slow (1-MHz) Components

As you can see from the block diagram see Figure 1-4, the slow side includes the computer's built-in sound circuitry, the second 128K of standard RAM, plus all the components necessary for the IIGS to communicate with peripheral devices. A major component here is a chip called *Mega II*.

Mega II contains all the circuitry necessary to produce the display modes needed by programs for earlier Apple IIs (low-resolution, high-resolution, and so on). Because this is no easy task, Mega II is virtually packed

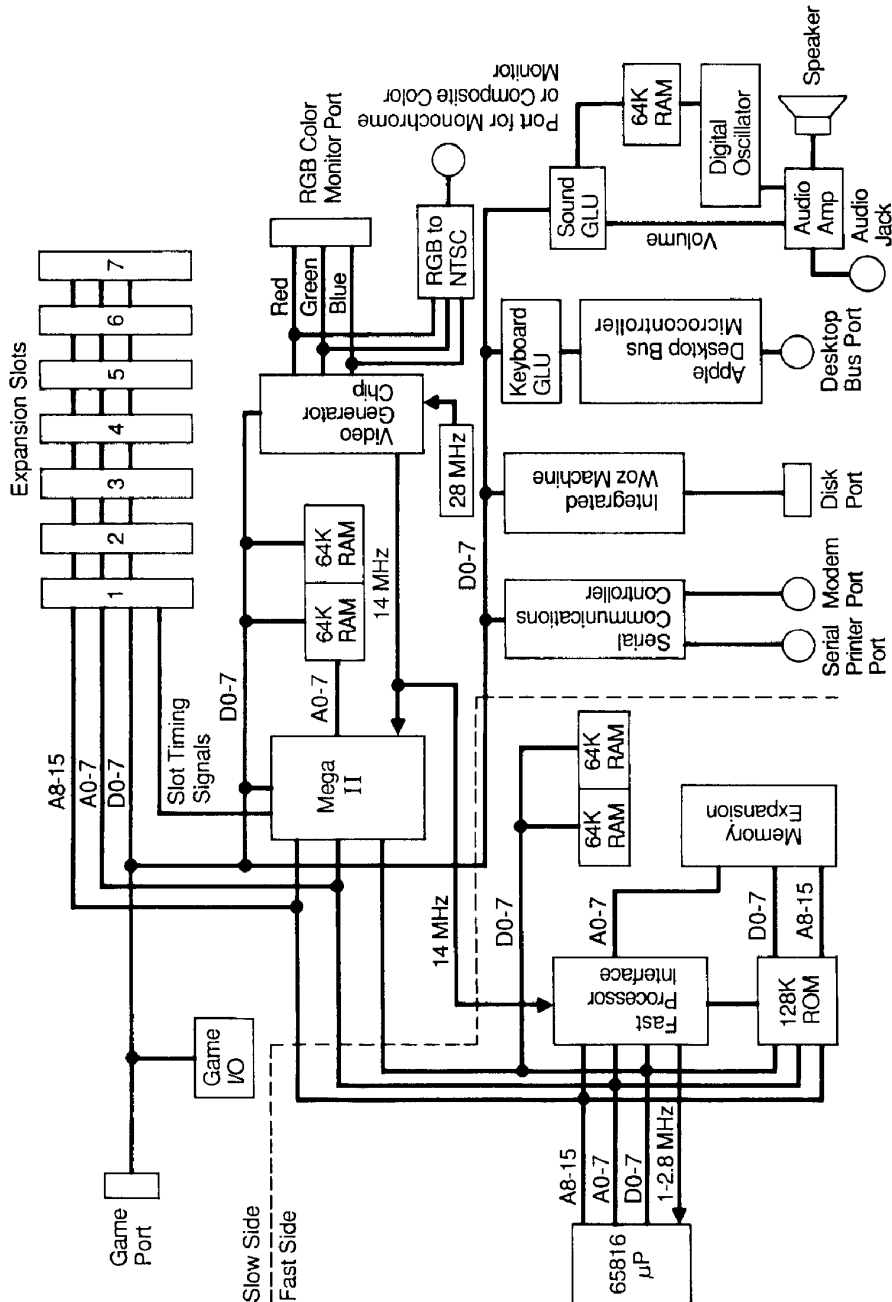


Figure 1-4

with components. In all, it contains the equivalent of about 3000 transistors and a 2K ROM that holds display characters. Mega II is, in fact, close to being an Apple II on a chip! Besides its display functions, Mega II also interacts with external devices plugged into the computer's expansion slots and rear-panel connectors (or ports).

The *video generator chip (VGC)* does the actual interfacing with attached display monitors. Its output lines are converted to red, blue, and green signals for RGB monitors and are combined into a single signal for monochrome and composite video monitors, such as home television sets.

The *serial communications controller (SCC)*, *integrated Woz machine (IWM)*, and *Apple desktop bus microcontroller (ADBM)* service the serial, disk, and desktop bus ports on the computer's rear panel. Both the disk and desktop port can handle a number of peripherals that are "chained" together. For example, the IIGS keyboard connects directly to the desktop bus port and the mouse is chained to it.

System Memory

As I mentioned earlier, the 65816 can address 16 megabytes of memory. Figure 1-5 shows how the Apple IIGS uses this space.

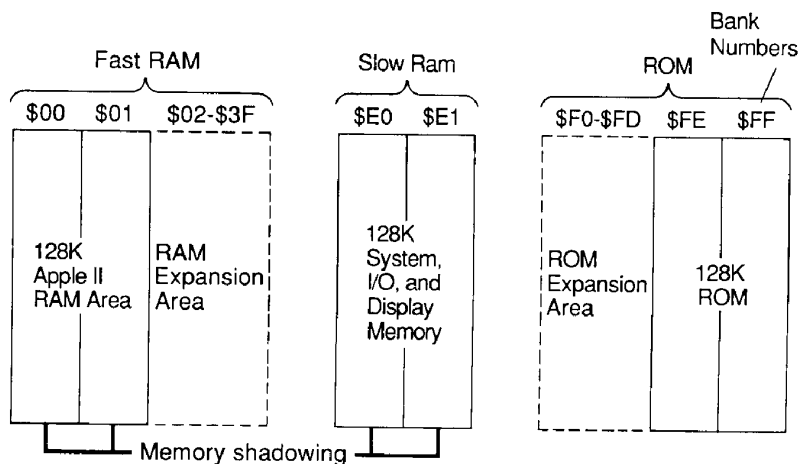


Figure 1-5

Read/Write Memory

The base model IIGS comes with 256K bytes of read/write memory, or *RAM*, consisting of four 64K banks: \$00, \$01, \$E0, and \$E1. (RAM, short for random access memory, is really a misnomer, since disk storage is also random access. It might just as well be called RWM, but it's too late to change the industry now.) Apple II programs use the first 128K. The system uses some of the second 128K for working storage, I/O (remember, the 65816 treats external devices just like ordinary memory), and the display. I'll discuss that second 128K, banks \$E0 and \$E1, shortly. Programs written for the Apple IIGS — that is, those that run in the 65816's native mode — can use about 176K of the 256K bytes available.

Banks \$02 through \$3F provide memory space for a RAM expansion card that can be plugged into the IIGS motherboard. In all, the 62 banks numbered \$02 through \$3F provide addressing space for an additional 3.875 additional megabytes of RAM, bringing the total RAM capacity to 4.125 megabytes. All of the expansion RAM is available to application programs; the system doesn't use it.

Memory Shadowing

Note that in Figure 1-5, banks \$E0 and \$E1 are labeled *slow RAM*, while the rest is *fast RAM*. Slow RAM operates at the Apple II's normal speed, 1 MHz, while fast RAM operates at the IIGS's normal speed, 2.5 MHz.

You're probably wondering how Apple II programs can run in banks \$00 and \$01, which is fast, 2.5 MHz RAM. The designers took care of this problem by using a technique called *memory shadowing*. When a program writes something into bank \$00 or \$01, the IIGS writes the same thing into the corresponding location in bank \$E0 or \$E1. Because the memory in \$E0 and \$E1 is synchronized to the video hardware, the instruction must execute at the slow speed. However, that only applies to write operations; the 65816 always *reads* from the affected areas of banks \$00 and \$01 at the faster speed.

To summarize, the trick is: only fast memory is read; both fast and slow memory are written. A write must be constrained to the 1-MHz speed because the operation isn't done until the slower component has been accessed.

Banks \$E0 and \$E1

Figure 1-6 shows a memory map of banks \$E0 and \$E1. Starting at the top of the figure, the reserved 1K is a working storage area for various IIGS

24 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

system software. There is another working storage area at address \$0C00. Starting at address \$0400 are the buffers that hold screen data for text and low-resolution graphics. The regular and double high-resolution buffers start at \$2000. Note that when super hi-res is active, the buffer extends from location \$2000 to location \$9FFF; that's 32K total. The last 8K of the free space is the memory that the Memory Manager manages. Finally, you encounter the I/O, banks 0 and 1, and language-cards areas.

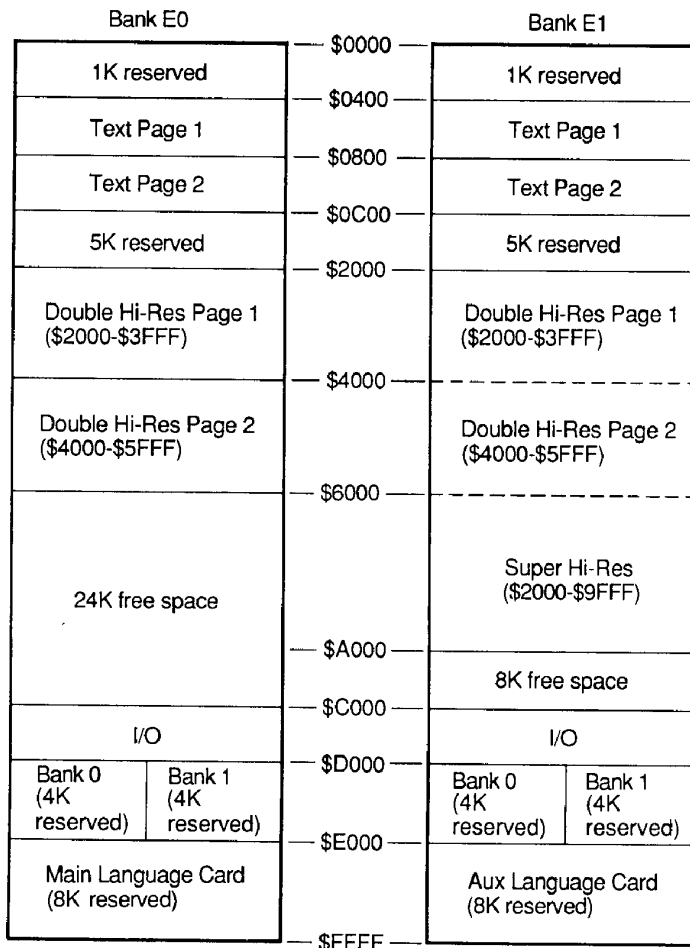


Figure 1-6

ROM

The ROM in banks \$FE and \$FF contains the following firmware:

1. Monitor
2. Control Panel menu
3. Mouse support
4. Appletalk
5. Support for modem and printer serial ports
6. Disk support
7. Front Desk Bus (FDB) support
8. Diagnostics
9. Desk Accessory Manager support
10. Tools

Programming the Apple IIGS

Like all microcomputers, the Apple IIGS has an internal operating system stored in read-only memory. This internal operating system, called the *Monitor*, holds the program that starts the computer when you switch the power on. It also holds programs that communicate with the keyboard, display information on the screen, and transfer data to the printer and whatever else is attached to your computer. The Monitor does these jobs by running miniprograms called *subroutines* contained in the ROM. There are many useful subroutines within the Monitor and, if you wish, you can use them to do common programming chores such as displaying a message on the screen or reading what the operator types at the keyboard.

In other Apple IIs, the Monitor subroutines are the only built-in aids available to the programmer. While the subroutines provide shortcuts for simple chores, they fall short in several areas. For example, the Monitor has no subroutines that multiply or divide numbers (functions that the 6502 and 65816 microprocessor instruction sets don't do), nor does it have subroutines that produce high-resolution graphics. The Monitor doesn't include these kinds of operations because they aren't needed to make the computer operate — and, after all, that's the Monitor's sole purpose of existence.

Thus, if you want to multiply, divide, or display graphics on an Apple II, II +, IIe, or IIfx, you must either find some software that will do the job for you or (gasp!) create the software yourself.

Fortunately, as an Apple IIGS programmer, you need not rely on the Monitor subroutines to help you do things. In fact, you may *never* use the Monitor, because the IIGS includes a large *Toolbox* that includes powerful programming aids called *tool sets*. Some of these tool sets are built into ROM, while others are fetched from the Apple IIGS System Disk. Each tool set is responsible for a particular variety of tasks. One set displays graphics, another handles sound generation, still another handles communication with the keyboard and mouse, and so on.

Just as a plumber's toolbox contains tools for working with pipes, and an electrician's toolbox has tools for working with wires, sockets, and fuses, each Apple IIGS tool set contains *tools* that relate to its specific area of responsibility. For example, the graphics tool set (called QuickDraw II) has tools that draw shapes on the screen, fill in areas with a specified color, and so on.

Quite simply, there is a tool for virtually every programming task one might want to do. In light of that, your assembly language programs will look quite different from programs people wrote for other Apple IIs. Instead of dozens (or hundreds or thousands) of assembly language instructions and calls to the Monitor, your programs will consist primarily of tool "calls," with just a sprinkling of assembly language instructions here and there.

Because the tools do most of the work, your programs will be much shorter, "cleaner," and easier to understand than equivalent programs written for other Apple IIs. They'll also take you less time to develop.

I'll discuss using tools later in the book, but in the next few chapters, I'll introduce the mechanics of developing assembly language programs and describe the 65816's instruction set. Read this material casually to get an overall understanding of it, but *don't* try to memorize every detail. In later chapters, where I describe actual programs, you will see how the microprocessor instructions and instructions to the assembler (or *directives*) fit in. In writing your own programs, you can always come back to these earlier chapters to look up specific details of an instruction or directive.

CHAPTER 2

Using an Assembler

Assembly language offers the best of two worlds. That is, it lets you write programs at the level the microprocessor understands, which helps ensure that they will be both fast and efficient. Yet assembly language doesn't force you to memorize a lot of numeric codes. Instead, you enter instructions as English-like abbreviations, then run an *assembler* program to convert the abbreviations to their numeric equivalents.

The program comprised of abbreviations is called the *source program*, while the numeric, microprocessor-compatible form of it is the *object program*. Thus, the assembler's job is to convert source programs you can understand into object programs the microprocessor can understand. It does much the same thing a compiler does in a high-level language such as BASIC, C, or Pascal.

There are several assembler software packages available for the Apple IIGS, but this book describes just one: Apple Computer's *Apple IIGS Programmer's Workshop* (or *APW* for short). The assembler within the *APW* is actually an enhanced version of the popular *ORCA/M* assembler from The Byte Works, Inc., so *ORCA* programmers will feel quite comfortable with it. In any case, the features the *APW* provides should be similar to those of any other IIGS-compatible assembler you might have, because they all deal with the same computer.

The *Programmer's Workshop* manual provides complete details, so I won't attempt to describe everything. Instead, I'll concentrate on the features you will probably use most often, and provide some summary tables for quick reference.

Developing an Assembly Language Program

Although assembly language programs look quite different from BASIC, C, or Pascal programs, one follows the same procedures to develop them. However, in assembly language the *mechanics* are more involved. There are seven steps in developing an assembly language program:

1. Define the task and design the program. This often requires drawing a *flowchart*, a road map of how the program should operate.
2. Type the program instructions into the computer using an *editor*, then save the program on disk.
3. Assemble the program using the *assembler*. This produces a disk file called an "object module." If the assembler reports errors, correct them with the editor and reassemble the program.
4. Convert the object module to an executable "shell load file" using the *linker*. Shell load files can be run from the *Programmer's Workshop* disk.
5. If the program is to be run from the System Disk's Program Launcher, redefine the shell load file as a "system load file."
6. Execute (run) the program.
7. Check the results. If they differ from what you expected, you must find the errors or "bugs." The *debugger* is handy for doing this.

If your program is short and simple, you can perform these steps quickly. But longer and more complex programs require more time on each step, especially defining the program. I discuss an efficient approach for developing programs under "Top-Down Program Design" at the end of this section.

Editor

Step 2 above refers to an editor. This is a program that lets you enter and prepare your program. You can use any word processor or editor program that can produce pure ASCII text — regular characters without any special control codes or formatting codes. One such program is *MouseWrite*, from Roger Wagner Publishing, Inc. If you don't have one of these programs, you can use the editor program that comes on the *Programmer's Workshop* disk (see the editor later in the chapter).

Assembler

The computer cannot execute the program you prepare with the editor. You must use the assembler to convert it into an *object* module the computer can understand.

Linker

The linker converts object modules into *shell load files*. A load file contains the numeric (object) form of the program in a form the system loader can load into memory. A shell load file can be run under the *Programmer's Workshop* by entering its name. The linker also does another important job: it combines two or more object modules — a main module and one or more subroutine modules — to form a load file.

Note that *you must run the linker for every program you write*, even those that have only one object module. If a program has only one module, the linker simply puts it in loadable form. If a program has two or more object modules, the linker combines them and makes the *result* loadable. Figure 2-1 illustrates the stages involved in editing, assembling, and linking a program.

If you want to be able to run the program from the System Disk's Program Launcher, there is one more job to do: you must redefine the shell load file as a *system load file*. This takes only a simple command that changes the file's "type."

Debugger

Unfortunately, most programs don't run exactly as expected the first time; they usually contain errors. The easiest kind of error to correct is a syntax error, such as accidentally typing *%E4* when you intended to type *\$E4*, or mistyping an *LDA* instruction as *LDQ*. Syntax errors are easy to spot because the assembler identifies them and issues an appropriate error message when it assembles the program.

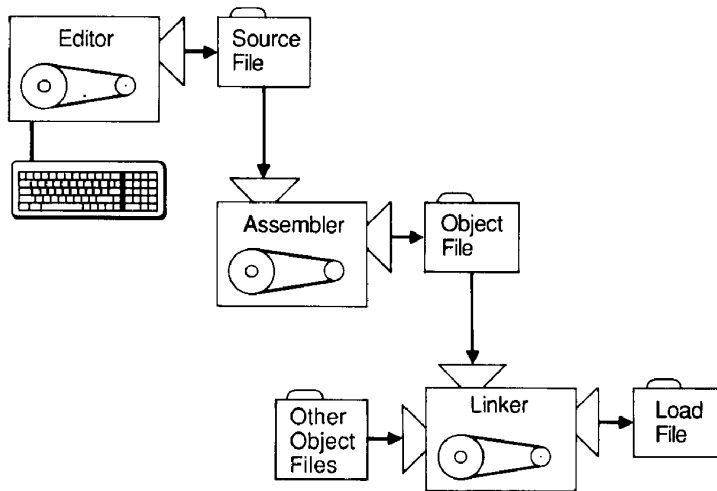


Figure 2-1

Errors that show up when you run the program are also sometimes easy to correct. For example, if the prompt “Please enter your name” shows up as “Please enter your *nabe*”, you know at once that there’s a typo in your program’s message string. Simply correct it with the editor and reassemble the program.

If syntax and typographical errors are the only kinds you encounter, you’re either a genius or living a charmed life. Most people occasionally get errors that are difficult to locate. The more minor errors just produce an incorrect result, such as a 26 where you expected a 140. Other errors “crash” the program, sending the computer off to a never-never land in which the screen goes blank. Here, your only recourse is to switch the power off.

When these kinds of things happen, programmers usually list the program and search for obvious errors. More often than not, their search is fruitless. Where does one go from here? Well, generally the best approach is to run the program one instruction at a time, and check each step of the way whether everything is going as expected. The utility that lets you “step” through programs is called the *debugger*.

Among other things, the debugger also lets you display and change values and stop the program at a specific point, to see how things are

proceeding. The debugger thus provides tools that help you to identify and correct errors in programs.

Top-Down Program Design

When creating a program on a computer, you are usually inclined to charge into it just as you would with pencil and paper, entering the first instruction, then the second, third, and so on, to the end. This “brute force” approach may work for programs that are short or simple, but generally it leads to errors and produces programs that are difficult to understand and even more difficult to update later. Thanks to the editing capabilities of word processors and editors, there is an easier, more reliable, and more efficient way to develop programs. It is called *top-down design*.

Top-down design simply means starting with general issues and progressing methodically to specifics. Outlines, tree diagrams, and flowcharts are all valid mechanisms for achieving the ends of top-down design.

One way to approach top-down design is to start with a plain-English outline of the program, then filling in the details gradually. The outline should be a series of lines that tell what steps you want the program to perform. For example, to develop a program that performs one of several tasks based on a choice the user makes from a menu, your outline might look like this:

```
; Display a menu of selections.
; Ask the user to choose.
; Read the user's selection.
; Do what the user has asked.
```

The semicolons here indicate that these lines represent comments rather than instructions. They do the same thing as REMs in BASIC.

From here, you can use the editor to insert instructions between the comment lines. Because each line defines a simple task, you can complete them individually and test each one before proceeding to the next one. That is, begin by inserting the first group of instructions (the ones that display a menu, in my example), then save the program on disk and assemble, link, and execute it. Executing this partially completed program tells you whether it is working correctly so far. If it isn't, debug it and try again. When the first part is working, proceed to the second part, then the third, and so on.

This may seem like a slow way to develop a program, but it has several advantages:

1. It forces you to plan the program in an orderly fashion.
2. The comment lines provide a certain amount of top-level documentation to the finished product.
3. It helps ensure that each step is working correctly before you proceed. This way, you know that any error was probably caused by something you did in the current group of instructions, not in a previous group.

Source Statements

Now that I have discussed the mechanics of developing programs, it's time to take a look at what can go into them. The source program you enter into the computer is a sequence of *statements* that are designed to perform a specific task. A source statement (a line in a program) can be any of five things:

1. A blank line.
2. A comment.
3. An assembly language instruction.
4. An assembler directive.
5. A macro call.

Assembly language instructions are shorthand for the 65816 microprocessor's instruction set. Some manuals refer to them as machine language instructions, because they tell the "machine" (the 65816) what to do. By contrast, assembler directives tell the *assembler* what to do (with the instructions and data you enter).

Macros are labeled groups of instructions. To run the entire group, you simply enter its name and specify any parameters it is designed to accept. You could think of a BASIC statement such as PRINT as a macro, because the interpreter or compiler must replace PRINT with several machine language instructions. (Remember, regardless of which language your program is written in, the microprocessor only understands machine language. Every program file must ultimately translate to machine language.) Macros are discussed in Chapter 5.

Assembly Language Instructions

Each assembly language instruction in a source program can have up to four *fields*, as follows:

```
{Label}      Mnemonic      {Operand}      {; Comment}
```

Of these, only the mnemonic field is always required. The label and comment fields are always optional. The operand field applies only to instructions that require an operand; otherwise, you must omit it. (I show the label, operand, and comment fields in braces to identify them as optional; *don't* type the braces in your programs.)

The label must start in column 1, but you can enter the other fields anywhere on the line, as long as you separate them with at least one space (or tab). An assembly language instruction that uses all four fields is:

```
SetCount      LDX      #4      ;Initialize count
```

Label Field

The label field assigns a name to an assembly language instruction, letting other instructions in the program refer to this instruction. Thus, labels in assembly language programs serve the same purpose as line numbers in BASIC programs.

The label must start in column 1 and cannot contain blanks. It must begin with an alphabetic character, *A* through *Z* — or *a* through *z*, the assembler doesn't distinguish between lowercase and uppercase. The remaining characters may consist of:

- The letters *A* through *Z* (or *a* through *z*).
- The numeric digits 0 through 9.
- The underscore character (`_`).
- The tilde character (`~`).

The assembler can process instruction labels of up to 255 characters long (!). However, because most printers produce 80-column lines, the practical limit is somewhat less than 80 characters, to provide for mnemonic and operand fields.

Because the assembler lets you enter various combinations of characters, most labels you can think of are acceptable. However, I recommend the following guidelines for selecting labels:

- Make the name as short as possible, while still being reasonable.
- Make the name easy to type without errors. The usual typing problems are several identical letters in a row (such as HHHH) and similar-looking characters (such as the letter O and the number 0, letter I and number 1, letter S and number 5, and letter B and number 8). There is no reason to invite typing errors; most of us make enough of them anyway.
- Don't use labels that can be confused with each other. For example, avoid using similar names like DataLoc and DateLoc. There's no sense tempting fate and Murphy's law.

Mnemonic (Opcode) Field

The mnemonic field (the first *m* in mnemonic is silent) contains the three-letter abbreviation for the instruction. For example, LDA is the abbreviation for the Load Accumulator instruction and JMP is the abbreviation for the Jump instruction. The assembler uses an internal table to translate each mnemonic into its numeric equivalent.

Many instructions require you to specify an operand as well as a mnemonic. For example, the JMP instruction must know where to jump. The mnemonic tells the assembler what type(s) of operand, if any, it should expect to find in the operand field.

Operand Field

The operand field tells the 65816 where to find the data it is to operate on. This will be either the data itself, a memory address, or a register name. The operand field is mandatory with some instructions and prohibited with others. (The addressing characteristics for each instruction in the 65816's instruction set are discussed in Chapter 3.)

The assembler differentiates constants in the operand field based on which prefix is used. A hexadecimal or binary number must be preceded by a \$ or % symbol, respectively, while a decimal number must be unprefixes. For example, the following instructions are equivalent:

```
LDA    #$E        ;Hexadecimal
LDA    #%11110    ;Binary
LDA    #14         ;Decimal
```

Text characters must be enclosed in single quotes ('). For example, *LDA #'Y'* loads the numeric code for capital letter Y into the accumulator.

Comment Field

Like a REM in BASIC, this optional field lets you describe statements in a program, to make the program easier to understand. You must precede a comment with either a semicolon (;), an asterisk (*), or an exclamation point (!) — most programmers use the semicolon — and separate it from the preceding field by at least one space or tab. The assembler ignores comments, but prints them when you list the program.

Put anything you want in a comment field, but to be useful, make comments describe what is happening in the program, not just restate the instruction. For example,

```
LDA    #0        ;Set result register to 0 to begin
```

is more meaningful than

```
LDA    #0        ;Move 0 into 1
```

You may also put a comment on a line by itself, to describe a block of instructions. To do this, enter a semicolon at the beginning of the line. The assembler recognizes ; as the start of a comment line and ignores whatever follows it.

Assembler Directives

Directives are commands to the assembler, rather than to the microprocessor. They can be used to set up subroutines, define symbols, reserve memory for temporary storage, and perform a variety of other housekeeping tasks. Unlike assembly language instructions, however, most directives generate no object code.

Directive statements can have up to four fields. They are:

```
{Label} Directive {Operand} {;Comment}
```

As the braces indicate, only the directive itself is always required. A label is mandatory with some directives and optional with the rest. The same applies to an operand. Of course, the comment is always optional. As with an assembly language instruction, you can put a directive anywhere on a line (except in column 1), but you must separate it from the label with at least one space or tab. Table 2-1 divides the commonly used directives into six groups: program control, file control, space allocation, equate, listing, and mode.

Program Control Directives

In the Apple IIGS, assembly language programs are divided into named groups called *code segments*. Every program has at least one code segment: the segment that holds the main program. That code segment may also contain subroutines that the main program uses. If you want to call a subroutine from some other segment, however, you must define it in a separate code segment.

Each code segment must begin with a *START* directive and end with an *END* directive. The *START* directive must be preceded by a label, which is the name of the segment. For example, a segment named *MainSeg* that includes a call to a subroutine named *Subr1* would have this general form:

```

MainSeg      START           ;Start of main segment
              . .
              . .
              JSR Subr1      ;Call Subr1
              . .
              . .
              END           ;End of main segment
Subr1        START           ;Start of subroutine segment
              . .
              . .
              END           ;End of subroutine segment
```

Again, this assumes that *Subr1* is a subroutine you want to share between segments. If it is used only by *MainSeg*, you could put it in *MainSeg*'s segment. (In that case, you wouldn't need the second *START* and *END*. *Subr1*

Table 2-1

Function	Format		
Program Control			
Start code segment	<i>label</i>	START	
End segment		END	
Define global entry point	<i>label</i>	ENTRY	
Start data segment	<i>label</i>	DATA	
Use data segment		USING	<i>label</i>
Enable/disable long accumulator and memory		LONGA	ON/OFF
Enable/disable long index registers		LONGI	ON/OFF
File Control			
Append source file		APPEND	<i>pathname</i>
Copy source file		COPY	<i>pathname</i>
Save object file		KEEP	<i>pathname</i>
Space Allocation			
Define storage		DS	<i>byte-count</i>
Define constant		DC	{repeat-count} <i>value(s)</i>
Set origin		ORG	<i>address</i>
Equate			
Equate	<i>name</i>	EQU	<i>expression</i>
Global equate	<i>name</i>	GEQU	<i>expression</i>
Listing			
List/don't list output		LIST	ON/OFF
List/don't list absolute addresses		ABSADDR	ON/OFF
List/don't list symbol table		SYMBOL	ON/OFF
Send output to printer/to screen		PRINTER	ON/OFF
Start new page		EJECT	
Print page number and header		TITLE	{'header'}
Mode			
Enable/disable 65C02 instructions	65C02		ON/OFF
Enable/disable 65816 instructions	65816		ON/OFF

would simply be the label of the first instruction in the subroutine.) The JSR here stands for Jump to Subroutine; it does the same thing as GOSUB in BASIC.

START labels are termed “global,” because instructions in any segment of the program can refer to them. By contrast, labels *within* code segments are “local.” Only instructions within that segment can refer to them. (This has a side benefit: it means that you can use the same label in more than one segment.)

However, you can also declare an instruction global, by preceding it with an *ENTRY* directive. The label on the ENTRY directive assigns a global name to the instruction that follows. As its name implies, the ENTRY directive is convenient for providing an alternate entry point to a subroutine. For example, in a subroutine of the following form:

```
Subr1  START
      . .
      . .
AltEnt  ENTRY
      . .
      . .
      END
```

your program could contain a JSR Subr1 instruction to start at the beginning or a JSR AltEnt instruction to start at the ENTRY point (AltEnt).

You can also set up *data segments* to hold constants and variables that are used by your code segments. Every data segment must begin with a *DATA* directive (rather than a START directive) and end with an END directive. To use a variable or constant in a data segment, a code segment must contain a *USING* directive that names the data segment. For example, the statement USING DS1 must precede the first instruction that refers to an item in the data segment named DS1.

The *LONGA* and *LONGI* directives specify whether the accumulator and index registers are 8 bits long (OFF) or 16 bits long (ON). ON is the default to both directives. You can use these directives outside of a code segment as well as within one. LONGA OFF and LONGI OFF are convenient in emulation-mode programs. Without LONGA OFF, for example, the instruction

```
LDA    #4
```

that is, load decimal 4 into the accumulator would make the assembler assemble 4 as a 16-bit number.

File Control Directives

File control directives specify a disk path name as the operand. *APPEND* is used to construct programs that are too large for the editor to handle all at one time. *APPEND* simply reads a file in from disk and tacks it onto the current program. You can specify any valid ProDOS path name for the *APPENDED* file. If it is on a different disk, the assembler will tell you to switch disks before continuing.

COPY does the same thing as *APPEND*, except *COPY* may appear anywhere within a program, while *APPEND* may only appear at the end. *COPY* is convenient for inserting a file of equates (description upcoming) in a program.

KEEP is used to assign a file name to an object module (which must have a different name than its parent, the source module). You can only use *KEEP* once, at the beginning of the program and preceding the first *START* directive. For example, in a program called *MYPROG.SRC*, you could enter *KEEP MYPROG*. To store the object module on a different disk or in a different directory, you must precede *MYPROG* with a complete path name. The *KEEP* directive is optional, by the way. You can also name the object module when you assemble the program (details later).

Space Allocation Directives

Many programs use locations in memory to hold variables, and the assembler provides two directives to reserve space for them. *DS* (Declare Storage) reserves a specified number of bytes without assigning a value to them, whereas *DC* (Declare Constant) both reserves bytes and gives them an initial value. The *DC* directive is commonly used to define integers, addresses, hexadecimal constants, binary constants, and character strings. Table 2-2 shows the formats for these data types.

Table 2-2

Type	Format
Integer	{repeat-count}I{size}'value1,value2...'
Address	{repeat-count}A'address1,address2,...'
Hexadecimal constant	{repeat-count}H'digit(s)'
Binary constant	{repeat-count}B'digit(s)'
Character string	{repeat-count}C'string'

Note that you can precede the type identifier with a *repeat-count* that tells the assembler how many times to repeat the data between the single quote marks. Note also that you can follow the integer specifier, *I*, with a *size* value. This specifies the size of the integer in bytes; it can range from 1 to 8. If you omit the size, the assembler makes each data item two bytes long. For example, the following directive stores six integers in memory. The first four are two bytes long (because size is omitted), while the last two are one byte long (size = 1).

```
TABLE DC 2I'4,5',I1'1,2'
```

If you displayed that portion of memory, you would see:

```
04 00 05 00 04 00 05 00 01 02
```

Here are examples of the other types, including the hexadecimal values they store in memory:

Directive	Memory values
DC A'Table1,Table2'	(16-bit addresses of Table1 and Table2)
DC H'01234ABCD'	01 23 4A BC D0
DC H'AAAA BBBB CCCC'	AA AA BB BB CC CC
DC B'1011 0110'	B6
DC C'Are you sure?'	41 72 65 20 79 6F 75 20 73 75 72 65 3F

(See Appendix B for the numeric values of the characters in the string.)

When you run your program, the system loader will store it at any convenient place in memory. However, you can specify the starting location yourself by putting an *ORG* (short for Origin) directive at the beginning of the program. For example, *ORG \$3000* will locate the program at \$3000.

ORG's operand can also include an asterisk (*) to indicate the current location. Thus, *ORG *+2* makes the assembler skip 2 bytes before storing the next instruction. This statement does the same thing as *DS 2*.

Equate Directives

The equate directives assign the value of an operand expression to a name. Thereafter, you can use the name anywhere you would normally use the expression. The first of these directives, *EQU*, is a "local" equate; its name can only be used in the segment in which it is defined. The other, *GEQU*,

is a “global” equate; its name can be used in any segment in the program. Global equates are normally defined in data segments rather than code segments. Some examples of equates are:

TWO	EQU	2
FOUR	EQU	TWO*TWO
K	GEQU	1024

The operand for an equate can also contain an instruction label. If it is the label of a direct (zero) page address or a long address (I’ll describe address types in the next chapter), it must precede the labeled instruction; if it is the label of an instruction in any other page, it must follow the labeled instruction. Note that you can also use the *ENTRY* directive to define a global label.

Listing Directives

The assembler automatically displays the names of your program’s segments as it assembles them. However, you can also make it list the numeric machine code for each instruction by entering a *LIST ON* directive at the beginning of the program. If you want to list only a portion of the program, you can shut off the listing with *LIST OFF*. These directives can also be applied selectively within a program. To list a subroutine, for example, precede it with *LIST ON* and follow it with *LIST OFF*.

Program listings include an address for each instruction, but they are given relative to the start of the segment. (Each segment starts at address 0.) To obtain addresses relative to the start of the program, enter an *ABSADDR ON* directive at the beginning. These so-called absolute addresses do not indicate where in memory the program will actually be loaded, because loading is the system loader’s job, not the assembler’s. However, you can easily add them to the starting load address (more on this later) to calculate actual or “effective” addresses. Having effective addresses comes in handy during debugging.

The assembler also generates an alphabetical list of all local symbols after each *END* directive and a list of all global symbols at the end of the program. The *SYMBOL* directive lets you turn symbol table generation ON or OFF. This speeds up the assembly slightly and saves paper.

The *PRINTER* directive determines whether the assembler listing is sent to the printer (*PRINTER ON*) or the display screen (*PRINTER OFF*). The default is OFF. The assembler assumes that you have an 80-column printer connected to IIGS.

EJECT makes the printer start a new page. This is handy for dividing a listing into logical groups — say, to put each subroutine on a page of its own or to separate symbol tables from listings.

The final listing directive, *TITLE*, prints the page number and (optionally) a header at the top of each page. You can use as many *TITLE* directives as you want. For example, you can enter a title for each subroutine, to give your listing a polished, professional look.

Mode Directives

The assembler is designed to work with 6502 and 65C02 programs, as well as those written for the 65816. The 65816 includes all the 6502 and 65C02 instructions and addressing modes, plus some additional ones of its own (details are upcoming in Chapters 3 and 4). The mode directives tell the assembler which microprocessor's instruction set to recognize. The first one, *65C02*, lets you tell the assembler whether to accept 65C02 instructions (65C02 ON) or only 6502 instructions (65C02 OFF). The first form is convenient for writing programs to run on an Apple //c; the second can be used to write Apple II, II+, and //e programs.

The second mode directive, *65816*, lets you tell the assembler whether to accept 65816 instructions and addressing modes (65816 ON) or only those for the 65C02 and 6502 (65816 OFF). The latter form is convenient if you are writing a program for earlier Apple IIs, because with it, you needn't worry about whether you have used a 65816 feature accidentally.

Advanced Directives

The assembler offers a variety of other directives (see Table 2-3), but most are only of use for advanced applications. If you're interested in any of these directives, refer to Apple's *Assembler Reference* manual. There are also some directives related to macros; they'll be discussed in Chapter 5.

Operators

Note: This is primarily a reference section. If you are a beginner, read it casually, then come back later if you need to look up details.

An *operator* is a modifier used in the operand field of an assembly language or directive statement. There are three kinds of operators: arithmetic, logical, and relational. Table 2-4 summarizes them.

Table 2-3

Function	Format		
Program Control Directives			
Define private code segment	<i>label</i>	PRIVATE	
Define private data segment	<i>label</i>	PRIVDATA	
Memory Designation Directives			
Align to a boundary		ALIGN	<i>number</i>
Reserve memory		MEM	<i>start, end</i>
Assembler Option Directives			
Generate IEEE format number		IEEE	ON/OFF
Set maximum error level		MERR	<i>level</i>
Set the most-significant bit for characters		MSB	ON/OFF
Specify case sensitivity		CASE	ON/OFF
Specify case sensitivity in object module		OBJCASE	ON/OFF
Listing Option Directives			
Expand DC statements		EXPAND	ON/OFF
Set comment column		SETCOM	<i>column-number</i>
Show instruction times		INSTIME	ON/OFF
List/don't list errors		ERR	ON/OFF
Miscellaneous Directive			
Rename opcodes (mnemonics)		RENAME	<i>original, new</i>

Arithmetic Operators

The arithmetic operators combine numeric operands and produce a numeric result. The most common arithmetic operators are those that add (+), subtract (−), multiply (*), and divide (/). The divide operator returns the quotient produced by a divide operation. For example,

```
Pi_Quot EQU 31416/10000
```

returns 3.

Finally, the bit shift operator (|) displaces a numeric operand to the left or right, depending on whether the specified bit count value is positive or negative. About the only time you need this capability is when you're setting up "masks" that you will apply to binary patterns in memory. For example, if you set up a mask with the statement

Table 2-4

Operator	Function
Arithmetic	
+	Format: value1 + value2 Adds <i>value1</i> and <i>value2</i> .
-	Format: value1 - value2 Subtracts <i>value2</i> from <i>value1</i> .
*	Format: value1 * value2 Multiplies <i>value2</i> by <i>value1</i> .
/	Format: value1 / value2 Divides <i>value1</i> by <i>value2</i> , and returns the quotient.
	Format: value count Shifts <i>value</i> by <i>count</i> bit positions. Shifts left if <i>count</i> is positive and right if it is negative.
Logical	
.AND.	Format: operand1 .AND. operand2 True (1) if both operands are nonzero; false (0) if either operand is zero.
.OR.	Format: operand1 .OR. operand2 True (1) if either or both operands are nonzero; false (0) if both operands are zero.
.EOR.	Format: operand1 .EOR. operand2 True (1) if either operand, but not both, is nonzero; false (0) if both operands are nonzero.
.NOT.	Format: .NOT. operand True (1) if the operand is zero; false (0) if it is nonzero.
Relational	
=	Format: operand1 = operand2 True (1) if the operands have the same value; false (0) if they have different values.
<>	Format: operand1 <> operand2 True (1) if the operands have different values; false (0) if they have the same value.
<=	Format: operand1 <= operand2 True (1) if <i>operand1</i> is less than or equal to <i>operand2</i> ; false (0) if <i>operand1</i> is greater than <i>operand2</i> .
>=	Format: operand1 >= operand2 True (1) if <i>operand1</i> is greater than or equal to <i>operand2</i> ; false (0) if <i>operand1</i> is less than <i>operand2</i> .

Table 2-4 (cont.)

Operator	Function
>	Format: <i>operand1</i> > <i>operand2</i> True (1) if <i>operand1</i> is greater than <i>operand2</i> ; false (0) if <i>operand1</i> is less than or equal to <i>operand2</i> .
<	Format: <i>operand1</i> < <i>operand2</i> True (1) if <i>operand1</i> is less than <i>operand2</i> ; false (0) if <i>operand1</i> is greater than or equal to <i>operand2</i> .

```
Mask EQU %10110010
```

the statement

```
Mask_Left_2 EQU Mask|2
```

sets up a new constant with the value %11001000. Similarly,

```
Mask_Right_2 EQU Mask|-2
```

sets up a new constant that has the value %00101100.

Logical Operators

Logical operators are so named because they operate according to the rules of formal logic, as opposed to the rules of mathematics. The rules of logic are formed with a series of true/false “if” statements that lead to a “then” conclusion. A typical example is “If A is true and B is true, then C is true.” In fact, this very statement has an assembly language counterpart in the **.AND.** operator.

.AND. tests two operands and returns a 1 (true) if both have a value other than zero (true). If either operand is zero (false), **.AND.** produces a result of 0 (false).

.OR. is somewhat more liberal. It produces a 1 (true) result if either or both operands are nonzero.

.EOR. is a variation on **.OR.** that returns a 1 (true) if either operand, but not both, is nonzero. In fact, the “but not both” exclusion is how **.EOR.** got its name; EOR is short for Exclusive-OR. Table 2-5 summarizes how **.AND.**, **.OR.**, and **.EOR.** operate.

Table 2-5

Operand #1	Operand #2	Result		
		.AND.	.OR.	.EOR
Nonzero	Nonzero	1	1	0
Nonzero	0	0	1	1
0	Nonzero	0	1	1
0	0	0	0	0

The final logical operator, *.NOT.*, tests only one operand. It produces a 1 (true) if the operand is zero, or a 0 (false) if the operand is nonzero. Note that *.NOT.* differs from the other three logical operators in that it is looking for a zero value rather than a nonzero value.

Relational Operators

Relational operators compare two numeric values or memory addresses and, like the logical operators, produce 1 if the relationship is “true,” or 0 if it is “false.” For example, if CHOICE is a predefined constant,

```
LDA  #CHOICE<20
```

assembles as either LDA #1 (if CHOICE is less than 20) or as LDA #0 (if CHOICE is equal to or greater than 20).

Because the relational operators can only produce two values (1 or 0), they are rarely used alone. Instead, they are usually combined with other operators to form a decision-making expression. For example, suppose you want the accumulator to contain 10 if CHOICE is less than 20 and to contain 5 otherwise. A statement that performs this task is:

```
LDA  #5+5*(CHOICE<20)
```

Here, if CHOICE is greater than or equal to 20, CHOICE<20 is “false,” and the assembler replaces it with 0. Since 5 times 0 is 0, the accumulator will receive 5 when you run the program.

You can even get fancier than that, by building more complex expressions. For example, suppose you want the accumulator to contain 10 if CHOICE is less than 20 but greater than 0 (i.e., CHOICE is between 1 and 19), and to contain 5 otherwise. This statement will do the job:


```
LDA  #5+5*( (CHOICE<20) .AND. (CHOICE>0) )
```

Recall that `.AND.` also produces a 1 (true) or 0 (false) result, so it is the perfect operator for letting the computer make range decisions like this one.

Entering, Assembling, and Running Programs

Since I haven't yet discussed the details of the 65816's assembly language instruction set (that's in Chapter 4), I can't expect you to write any useful programs at this point. Thus, I will provide a simple program that sounds a tone through the speaker inside the Apple IIGS.

The details of the program are unimportant, however. The main point is that you will learn how to enter a program into the computer, assemble it, produce a load module (i.e., an executable program file), and execute it. This exercise will give you hands-on experience with the basic steps, and should help you proceed with confidence through more complex material in the rest of the book.

Starting the *Apple IIGS Programmer's Workshop*

The *Apple IIGS Programmer's Workshop* (APW) disk is bootable, so you can start with it directly. You may want to do this to create a program that you will run in some later session. However, the APW disk does not contain all the tool sets your programs may need. These sets *are* contained on the Apple IIGS System Disk, however, so you should start using the following procedures:

1. Insert the System Disk into your main disk drive and turn the computer on. (If it is already on, insert the System Disk and press Ctrl-OpenApple-Reset.)
2. When the Program Launcher screen appears, replace the System Disk with the APW disk and click the mouse on *Disk*. The Program Launcher loads the APW disk (volume name /APW/) and lists its contents.
3. Run the highlighted file, APW.SYS16, by clicking on *Open*. This makes the APW prompt, `#`, appear.

You'll learn what to do next in a moment, but first here's the example program that beeps the speaker.

A Simple Speaker-Beeping Program

Every Apple II has a location in memory that controls the computer's internal speaker. Whenever the microprocessor accesses this location, by either reading from it or writing to it, the speaker emits a "click" sound. A single click isn't very exciting, but if you make the speaker click fast enough, it sounds like a continuous tone, or in some frequency ranges, like a musical note. The program I am about to present reads the contents of the speaker location repeatedly to produce a "beep." In other Apple II models, the speaker "occupies" location \$C030 of bank 0; in the Apple IIGS, it occupies location \$C030 of bank \$E0 (and is copied, with shadowing, into \$00C030).

Example 2-1 lists the speaker-beeping program. This program consists of two loops. The outer loop, which starts at *BeepIt*, controls the *duration* of the beep — how long it lasts. The inner loop, which starts at *Loop2*, controls the beep's *frequency* — the rate at which the speaker gets clicked. The faster the speaker is clicked, the higher pitched the tone will be.

Briefly, here's what the lines in the program do:

- The *keep beep* directive at the beginning tells the assembler what name to give the object file. The linker will also assign this name to the load file it produces.
- The *Beep START* directive marks the beginning of the program's only code segment. (The final *END* directive marks the end of this segment.)

Example 2-1

```
; BEEP beeps the speaker by repeatedly accessing the speaker
; location in bank $E0.

        keep  beep           ;Name object and load files

Beep     START
speaker  equ    $E0C030      ;This is the speaker location

        ldy    #1000         ;Y is outer loop (duration) counter
BeepIt   lda    speaker       ;Beep the speaker
        ldx    #1000         ;X is inner loop (frequency) counter
Loop2    dex          ;Decrement X by 1
        bne    Loop2         ;Loop until X is 0,
        dey          ; then decrement Y by 1
        bne    BeepIt        ;If Y is not 0, go beep the speaker
        rti             ;Otherwise, if Y is 0, exit
        END
```

- The *speaker* EQU directive simply allows me to use the word “speaker” in instructions where I would otherwise use the potentially cryptic number \$E0C030.
- The first instruction in the program, *ldy #1000*, loads decimal 1000 into the Y register. The program will repeatedly decrement Y by 1 until it contains 0. At that point, it will exit back to the calling program; the Programmer’s Workshop “shell,” in this case. Thus, Y determines the duration of the beep. (The value 1000 is arbitrary. I have used it only because it produces a beep that lasts long enough to be heard easily.)
- The instruction *lda speaker* at BeepIt clicks the speaker once. It loads the contents of the locations \$E0C030 and \$E0C031 into the accumulator, but the value it has obtained is inconsequential. I simply want to access the speaker location somehow, and this does the job.
- The *ldx #1000* loads decimal 1000 into the X register.
- The *dex* at Loop2 decrements (decreases) the X register by 1.
- The *bne Loop2* instruction (where *bne* stands for Branch if Not Equal to 0) tests what happened when X was decremented. If X contains anything other than 0, the processor transfers to the preceding *dex* at Loop2; if X contains 0, the processor “drops through” to the next instruction. Hence, this two-instruction loop makes the processor wait until *dex* has been executed 1000 times. The wait interval determines the frequency of the tone in that it controls how often the speaker location is accessed. (As before, the value 1000 is arbitrary. I could have used a lower number to produce a higher-pitched tone or vice versa.)
- The instructions *dey* and *bne BeepIt* complete the outer loop, the one that determines how long the beep lasts. With each pass through the loop, Y gets decremented by 1. As long as it contains a nonzero value, the processor “branches” to the instruction at BeepIt, to click the speaker. When Y reaches 0, the processor executes the *rtl* (Return Long) instruction, which makes it exit the program.

Entering the Program

Now it’s time for you to enter the example program into the computer. Start the computer using the *Programmer’s Workshop* disk, or reboot by pressing

Control-OpenApple-Reset. When the *Workshop* prompt (#) appears, enter the command:

```
asm65816
```

This tells the editor to format the program you are about to enter as an assembler *source code* file, as opposed to a text file or some other kind of file.

Now, start the editor by entering

```
edit beep.src
```

where *beep.src* (for beep.source) is the name of the new program. You're actually telling the editor to load BEEP.SRC from disk, but since that file does not yet exist, the editor starts with a blank screen. It shows a rectangular cursor at the upper left-hand corner and a format line and status line at the bottom.

The *format line* shows a dot for each column position and a caret symbol (^) for each tab position. The editor provides tabs at columns 10, 16, 41, 48, 56, 64, 72, and 80 (the end of the line) automatically. The first three tabs — at columns 10, 16, and 41 — are convenient for entering the mnemonic, operand, and comment fields (respectively) of assembly language instructions.

The *status line* reports the position of the cursor (it is at "Line: 1" and "Col: 1" initially), the editor mode (EDIT, to begin), and the name of the file being edited (BEEP.SRC, in this case).

Now, enter the source program shown in Example 2-1, line by line, using the Tab key to move between fields and Return to start a new line. If you finish without omitting or mistyping anything, press *Control-Q* to leave or "quit" the editor (see "Leaving the Editor" in this chapter). Otherwise, if you made some mistakes along the way (most of us do), simply go back and correct them.

Correcting Typing Errors

The editor provides a variety of useful commands for manipulating text. Table 2-6 summarizes the ones you will probably use most often.

Note the following points about these commands:

1. Pressing Return from anywhere on a line moves the cursor to column 1 of the next line. This differs from most word processors, where pressing Return moves any remaining characters to a new line.

Table 2-6

(**Note:** The abbreviation *OA* represents the Open Apple key.)

Quit Command	
Quit (Exit to the Editor Menu)	OA-Q or Ctrl-Q
Cursor-Moving Commands	
Bottom of Screen/Page Down Moves the cursor to the bottom of the screen, at the current column position. If it is already at the bottom, the screen scrolls down one page (22 lines).	OA-DownArrow or Control-OA-J
Cursor Down	DownArrow or Control-J
Cursor Left	LeftArrow or Control-H
Cursor Right	RightArrow or Control-U
Cursor Up	UpArrow or Control-K
End of Line	OA-> or OA-. (period)
Start of Line	OA-< or OA-, (comma)
Tab	Tab or Control-I
Tab Left	OA-A or Control-A
Top of Screen/Page Up Moves the cursor to the top of the screen, at the current column position. If it is already at the top, the screen scrolls up one page (22 lines).	OA-UpArrow or Control-OA-K
Word Left (Previous Word)	OA-LeftArrow or Control-OA-H
Word Right (Next Word)	OA-RightArrow or Control-OA-U
Insert Commands	
Insert Line Inserts a blank line ahead of the current line. The cursor may be anywhere on the line when you issue this command.	OA-B or Control-B
Insert Space Inserts a space at the current cursor position by moving the remaining characters one column to the right.	OA-Spacebar
Delete Commands	
Delete Block Puts the editor in "select" mode, where it highlights text as you move the cursor. Pressing Return removes the highlighted block from the screen; pressing Esc cancels the delete operation. <i>Note:</i> You cannot Undo the work of the Delete Block command.	OA-Delete
Delete Character	OA-F or Control-F
Delete Preceding Character	Delete or Control-D

Table 2-6 (cont.)

Delete Commands (cont.)	
Delete Line The cursor may be anywhere on the line.	OA-T or Control-T
Delete to End of Line	OA-Y or Control-Y
Delete Word The cursor may be anywhere within the word.	OA-W or Control-W
Remove Blank Lines If the cursor is on a blank line, this command deletes that line and any others up to the next nonblank line.	OA-R or Control-R
Undo (Restore) Last Deletion Restores text deleted by any of the preceding commands except Delete Block.	OA-Z or Control-Z
Search and Replace Commands	
Search Down (Forward)	OA-L
Search Up (Backward)	OA-K
Search Down and Replace	OA-J
Search Up and Replace	OA-H
Block Move and Copy Commands	
Cut Puts the editor in "select" mode, where it highlights text as you move the cursor. Pressing Return removes the highlighted material from the screen and stores it in a temporary disk file called SYSTEMP. Pressing Esc cancels the cut operation.	OA-X or Control-X
Copy Puts the editor in "select" mode, where it highlights text as you move the cursor. Pressing Return copies the highlighted block into a temporary disk file called SYSTEMP. Pressing Esc cancels the copy operation.	OA-C or Control-C
Paste Inserts the previously cut or copied text at the cursor position, by reading the contents of SYSTEMP.	OA-V or Control-V
Tab-Changing Command	
Set or Clear a Tab If there is a tab stop at the cursor position, this command removes it. If there is no tab stop at the cursor position, it sets one.	OA-Tab or Control-OA-I

2. The editor always starts in overstrike mode, replacing existing text with characters you type. However, you can put it in insert mode by pressing Control-E or OA-E (the status line shows *Mode: EDIT*

INSERT). Here, the editor inserts typed characters at the cursor position and shifts the remaining characters on the line to the right. To return to overstrike mode, press Control-E or OA-E again.

3. When you delete a character, line, or word, the editor saves it in an “Undo buffer” in memory. The Undo buffer acts like a barrel, where each newly-deleted unit is placed on top of the unit that was last deleted. The Undo (Control-Z) command removes a unit from the top of the buffer and inserts it at the cursor position. Thus, by doing successive Undos, you can restore everything you deleted!

As you can see, the editor’s built-in power and flexibility make it more closely resemble a word processor than an ordinary “editor.”

Leaving the Editor

When you finish editing a program, press Control-Q or OA-Q to leave the editor. This produces the editor menu, which looks like Figure 2-2.

Now, to save the program on disk, press S for “Save to the same name” (i.e., save with the name BEEP.SRC). The editor writes the program to disk and shows “Writing . . .” on the screen. When the *Enter selection:* prompt reappears, leave the editor entirely by pressing E for “Exit without updating.” This makes #, the APW system prompt, reappear.

To make sure the program’s source file is really on the disk, issue a

```
File name: BEEP.SRC

(R) Return to editor.

(S) Save to the same name.

(N) Save to a new name.

(L) Load another file.

(E) Exit without updating.
```

Enter selection:

Figure 2-2

catalog command. The entry for the example program should have the *Name* BEEP.SRC, the *Type* SRC (for assembler source file), and the *Subtype* ASM65816.

Assemble, Link, and Run Commands

Now that the source file is on disk, you can use the commands the *Workshop* provides to assemble, link, and run it. Table 2-7 shows the commands you will probably use most often. To keep things simple, I have omitted some command parameters that are rarely used. Refer to the APW manual for complete details.

Table 2-7

Language Commands
<p>ASM65816 Lets the language to 65816 assembler source, thereby informing the editor to produce a SRC type file.</p>
<p>CHANGE <i>pathname language</i> Changes the language type of a source (SRC) or text (TXT) file. For example,</p>
<p>CHANGE MYFILE.SRC ASM65816</p> <p>changes MYFILE.SRC to the assembler source code type, ASM65816. This is handy if the editor was set to the wrong type (say, EXEC) when you created a file.</p>
<p>EXEC Sets the language to EXEC. EXEC files are used to store a list of <i>Workshop</i> commands. You run these commands by entering the name of the file.</p>
Edit Command
<p>EDIT [<i>pathname</i>] EDIT loads a specified text file or assembler source file into the editor. If the file does not exist, the editor starts with a blank screen and assigns the specified filename to it.</p>
Display Commands
<p>CATALOG [<i>pathname</i>] Displays the directory of the default disk (CATALOG) or subdirectory (CATALOG <i>pathname</i>) you specify. You can also use the = wildcard character in filenames. For example,</p>

Table 2-7 (cont.)

Display Commands (cont.)

CATALOG /MYFILES/P=

displays all the files on the MYFILES disk that begin with the letter P. You may abbreviate CATALOG as CAT.

HELP [*command-name*]

Displays a descriptive summary of an APW command. For example, *HELP DELETE* summarizes the DELETE command. If you omit the command name, HELP lists the names of all available commands.

TYPE [+N] *pathname* [*startline* [*endline*]] [>*device*]

Lists the contents of a text, source, or EXEC file. The parameters are:

- +N Makes APW number the lines.
 - pathname* The pathname of the file.
 - startline* The number of the first line to be listed. Omitting this parameter causes the entire file to be listed.
 - endline* The number of the last line to be listed. Omitting this parameter causes all lines between *startline* and the end of the file to be listed.
 - >*device* Sends the listing to an output device (e.g., >.printer). Omitting this parameter sends the listing to the screen.
-

Assembly and Link Commands

ASML [+L/-L] [+S/-S] *sourcefile* [KEEP = *outfile*]
 [NAMES = (*seg1* [*seg2* [. . .]])]

Assembles and links a source file. The optional parameters are:

- +L/-L Makes the assembler produce (+L) or omit (-L) a source listing. -L is the default.
- +S/-S Makes the assembler produce (+S) or omit (-S) an alphabetical listing of all global references in the object module. -S is the default.
- KEEP If your source file contains no KEEP directive, you must use this parameter to specify the name of the output (object) file. For a one-segment program, the object file is named *outfile.ROOT*. For a multi-segment program, the first (main segment) is stored in *outfile.ROOT* and the remaining segments are stored in *outfile.A*.
- NAMES Causes the assembler to assembled only the specified segments. The object code for these segments are stored in a file named *outfile.B*.

ASMLG [+L/-L][+S/-S] *sourcefile* [KEEP = *outfile*]
 [NAMES = (*seg1* [*seg2* [. . .]])]

Similar to ASML, except ASMLG (for "Assemble, Link, and Go") runs the program after linking it.

ASSEMBLE [+L/-L][+S/-S] *sourcefile* [KEEP = *outfile*]
 [NAMES = (*seg1* [*seg2* [. . .]])]

Assembles a source file, but does not link it. Hence, ASSEMBLE does not produce a load module. To link ASSEMBLED modules, use the LINK command.

Table 2-7 (cont.)

Assembly and Link Commands (cont.)

LINK [+S/-S] *objectfile* [KEEP=*outfile*]

or

LINK [+S/-S] *objectfile1 objectfile2 . . .* [KEEP=*outfile*]

Links object modules to create an APW shell load file. The first form of the command is used to link object modules that have the same name (e.g., MUSIC.ROOT and MUSIC.A). The second form is used to link modules that have different names (e.g., to link modules named TUNE with those named MUSIC). The +S/-S parameter is as described for ASML. The others are:

objectfile The pathname (minus filename extension) of the object modules to be linked. All modules to be linked must have the same filename (except for extensions) and must be in the same subdirectory.

For example, if the program MUSIC consists of the object modules MUSIC.ROOT, MUSIC.A, and MUSIC.B, all located in directory /MYFILES, use /MYFILES/MUSIC for *objectfile*.

KEEP Use this parameter to specify the name of the load file. **Warning:** If you omit KEEP, the linker will not produce a load file.

objectfilen You can use a single LINK command to link object files having different names into one load file, by giving their pathnames, minus filename extensions.

For example, suppose /MYFILES contains the object modules for a program called MUSIC (say, modules MUSIC.ROOT, MUSIC.A, and MUSIC.B) and the modules for another program called TUNE (TUNE.ROOT and TUNE.A). To link TUNE's modules with MUSIC's, specify

/MYFILES/MUSIC /MYFILES/TUNE

for *objectfile*.

File and Directory Commands

COPY [-C] *pathname1* [*pathname2*]

Copies a file to a different subdirectory, or to a duplicate file with a different name. The parameters are:

-C Normally, if a file named *newpath* already exists, APW asks if you want to replace it. The -C option lets you copy without the prompt.

oldpath The pathname of the file to be copied. Wildcards may be used to copy multiple files. If *oldpath* is a directory, COPY copies the directory and any subdirectories and files in it.

newpath The pathname of the copy. If you omit the pathname, the file is copied to the current directory.

CREATE *pathname*

Creates a new subdirectory with the specified pathname.

DELETE *pathname*

DELETE deletes the specified file. It can also delete a multiple files if you use a = wildcard in the pathname.

Table 2-7 (cont.)

File and Directory Commands (cont.)

ENABLE D[N][B][W][R] *pathname*

ENABLE enables one or more of the access attributes of a ProDOS file. It is generally used after a FILETYPE command, to guarantee the file has the attributes you want; in most cases, you enable all the attributes by specifying DNBWR.

FILETYPE *pathname filetype-abbrev*

Changes the ProDOS 16 "filetype" of a file. The filetypes are:

Abbreviation	File Type
BIN	ProDOS 8 binary load file
CDA	Classic desk accessory
DIR	Directory
EXE	Shell load
LIB	Library
NDA	New desk accessory
OBJ	Object
S16	ProDOS 16 system load file
SRC	Source
STR	Startup load
SYS	ProDOS 8 system load file
TOL	Toolkit load
TXT	Text

The *Programmer's Workshop* linker produces a shell load (EXE) file. To convert it to a ProDOS 16 system load (S16) file, enter:

filetype *pathname* S16

INIT *device* [*name*]

Formats a disk as a ProDOS volume. Here, *device* is the device number of the drive containing the disk to be formatted and *name* is the volume name for the disk. If you omit *name*, INIT names the disk BLANK.

MOVE *oldpath newpath*

Moves a file from one location to another; it does the same thing as a COPY command followed by a DELETE. The parameters are the same as for COPY.

PREFIX *directory*

Tells the *Programmer's Workshop* that any pathnames you enter are to be found in the specified *directory*. For example,

PREFIX /MYPROGS/MY.MACROS

tells it to look for files in the MY.MACROS directory of the disk named MYPROGS.

RENAME *oldpath newpath*

Changes the name of a file. The parameters are:

oldpath The pathname of the file to be renamed or moved.
newpath The pathname to which *oldpath* is to be changed or moved.

Table 2-7 (cont.)

File and Directory Commands (cont.)	
SHOW [LANGUAGE] [LANGUAGES] [PREFIX] [TIME] [UNITS]	
Lists information about the system in one or more of the following categories:	
LANGUAGE	Lists the current default language.
LANGUAGES	Lists the available languages.
PREFIX	Lists the current subdirectories to which the ProDOS 16 prefixes are set.
TIME	Lists the current date and time.
UNITS	Lists the available units, including device names and (for disks) volume names.
Debugger Command	
DEBUG	
Starts the debugger utility.	

The commands are divided into six groups: language, edit, display, assembly and link, file and directory, and debugger. You have already used the ASM65816 language command and the EDIT editor command. As Table 2-7 shows, the Programmer's Workshop provides assembly and link commands that let you do as much or as little as you want at one time. That is, you can:

- Assemble a file and link it later,
- Assemble and link it immediately (ASML), or
- Assemble, link, and execute it (ASMLG).

If you're an optimist, you may be inclined to assemble, link, and execute a program directly. You won't lose anything by doing that, because both the assembler and linker will stop if it can't proceed due to errors.

At this point, you should assemble, link, and execute the example program, *BEEP.SRC*, by entering:

```
ASMLG + L BEEP .SRC
```

(ASMLG is short for "Assemble, Link, and Go.") The assembler reads the source file from disk, then assembles it.

When the assembly process finishes, the listing shown in Figure 2-3

appears on the screen. (Well, not exactly; I cheated a little. Since the screen is only 80 columns wide, the comments “wrap” onto a new line. I omitted the comments from the figure so I could show each statement on a single line.)

Unfortunately, the listing scrolls by so fast that it’s virtually unreadable. To stop the scrolling temporarily, and to resume after it’s stopped, press any key.

Another solution — and probably a better one — is to make the assembler send its listing to your printer, rather than to the screen. To do this, insert a *PRINTER ON* directive at the beginning of your program, ahead of the *KEEP* statement. This assumes, of course, that you have configured the Apple IIGS to work with your particular printer and that the printer is “on-line.”

The Apple IIGS assumes that you have a serial interface card in slot 1 that operates with a specific set of parameters (9,600 baud, 8 data bits, 1 stop bit, no parity, and so on). If you have a serial printer with different

```

0001 0000
0002 0000
0003 0000
0004 0000
0005 0000
0006 0000
0007 0000
0008 0000
0009 0000 A0 E8 03
0010 0003 AF 30 C0 E0 BeepIt lda speaker
0011 0007 A2 E8 03      ldx #1000
0012 000A CA          Loop2 dex
0013 000B B0 FD      bne Loop2
0014 000D 88          dey
0015 000E D0 F3      bne BeepIt
0016 0010 6B          rti
0017 0011          END

17 source lines
0 macros expanded
0 lines generated
    
```

Figure 2-3

parameters, change the settings using the Control Panel's *Printer Port* option. If you have a parallel printer attached to slot 1, switch the slot setting to "Your Card" using the Control Panel's *Slots* option.

For this example and other short programs, you probably won't care about the listing. But when you're debugging long or complex programs, and trying to find that elusive "bug" that's making things go haywire, the listing can be quite useful — indeed, at times, indispensable. Therefore, it's worth spending a little time examining exactly what's being shown. The listing shows the following:

- The leftmost column shows the line numbers in decimal. (Except for instruction operands, the rest of the columns show hexadecimal numbers exclusively.)
- The second column shows where the object code for the statement (the instruction or directive) will be stored in memory *relative* to the starting address of the code. For example, if the system loader stores the program starting at location \$800 of bank 1, the first instruction, *ldy #1000*, will be stored at location \$800 (i.e., \$800 + offset \$0000); the next instruction, *lda speaker*, will be stored at \$803 (i.e., \$800 + offset \$0003); and so on.
- The third column shows the hexadecimal byte value that will be stored in memory for that particular instruction or assembler directive.

For an instruction, the byte represents the instruction's *opcode* — the numeric representation of the mnemonic and the addressing mode it uses. For example, *ldy*, which uses immediate addressing, is translated to the value \$A0. (Appendix C summarizes the opcodes for the 65816's instruction set.)

For an assembler directive that generates data (none of the directives in my program do), the byte in the third column represents the first value that the directive stores in memory. For example, the Declare Constant directive *dc C'H* stores a byte value of \$48 in memory, the ASCII code for the "H" character. (Appendix B summarizes the ASCII character set.)

- The remaining numeric columns show the operand values for each instruction that uses an operand. Some instructions take a 1-byte operand, others take a 2-byte or 3-byte operand, and a few, such as *dex*, *dey*, and *rtl*, take no operand at all.

For example, the operand “1000” for the *ldy* gets assembled into a 2-byte number, \$03E8. Note that on the listing \$03E8 appears as “E8 03”! This reflects the low-byte/high-byte order in which the 65816 stores numbers in memory. Similarly, the value of “speaker,” \$E0C030, is stored in low-byte to high-byte order — 30, then C0, then E0. Remember this arrangement when you’re viewing data in memory.

- The remaining columns of the listing are, of course, the source program that you entered using the editor.

The lines below the listing provide additional information about the assembly. The first, “17 source lines,” tells you that there are 17 lines in the source program (including blank lines), while “0 macros expanded” and “0 lines generated” refer to macros (a concept explained in a later chapter). There are no macros in this program; hence, the zeros.

When the linker finishes, it displays some information about the segments in the program (this has only one segment, *Beep*) and shows:

```
There is 1 segment, for a length of $00000011 bytes.
```

This says that the program is 17 (\$11) bytes long. (The 8-digit length field suggests that the linker can handle exceptionally large programs!)

Finally, the program is loaded into memory and run. The beep sounds for about three seconds and then the *#* prompt reappears. CATALOGing the disk at this point will reveal three files for your program:

- BEEP.SRC — The source file you created using the editor.
- BEEP.ROOT — The object file that the assembler produced.
- BEEP — The load file that the linker produced.

BEEP.SRC, BEEP.ROOT, and BEEP have the types *SRC* (for source), *OBJ* (for object), and *EXE* (for shell load), respectively.

If everything went smoothly and you heard the beep, you have successfully entered, assembled, linked, and run the program (congratulations!). Otherwise, if you received any error messages, you must edit the source program, then give another ASMLG command.

Shell Load Files and System Load Files

You can run the shell load (EXE) file that the linker has produced by entering its pathname from the `#` prompt. However, to run it from the System Disk's Program Launcher, you must convert it to ProDOS 16 system load file format. To do this, enter the command:

```
filetype pathname sl6
```

After that, cataloging the disk will reveal that your program has a "Type" of S16, which identifies it as a ProDOS 16 system load file.

Automating the Assembly Process

If your program is long or complex (or even short, but error-ridden), you will probably have to assemble and link it many times. Typing the same ASML command over and over is pure drudgery, and if you're fumble-fingered like I am, you often wind up with the added monotony of retyping commands. Fortunately, there is a way to automate the process: put your commands in an *EXEC* (Executive) file.

An EXEC file contains a list of *APW* commands that the *Wordshop's* shell program will execute when you enter the file's pathname. For example, suppose you want to assemble and link a program called MYPROG.SRC, then convert the linker's output to a system load file. An EXEC file that does this would contain:

```
asml myprog.src keep=myprog
delete myprog.root
filetype myprog S16
```

Here, the *keep* on the first line indicates that there is no KEEP directive in the program. Note the command to delete MYPROG.ROOT. Once you have linked a program, its object file is no longer needed; you can discard it.

To create an EXEC file, enter **exec** from the `#` prompt (that changes the active language to EXEC), then start the editor with a command of the form **edit *pathname***. EXEC files are, by convention, given the extension BUILD, so you may enter, say, **edit myprog.build**.

Once in the editor, enter the commands that belong in your EXEC file, then save it to disk. To run the file (i.e., to perform the commands in it), enter its name from the `#` prompt; e.g., enter **myprog.build**.

Multisegment Programs

The example program contains only one segment, *Beep*. However, if your program is large, you should divide it into functional segments. When the assembler assembles a multisegment program, it puts the first (main) segment in an object file that has the extension *ROOT* and puts all other segments in a file that has the extension *A*. For example, if *MYPROG.SRC* contains segments named *Main*, *Init*, and *DoIt*, the assembler would store *Main* in *MYPROG.ROOT* and store *Init* and *DoIt* in *MYPROG.A*.

The linker handles multiple object files automatically. If you enter an *ASSEMBLE*, *ASML*, or *ASMLG* command for *MYPROG.SRC* or a *LINK* command for *MYPROG*, the linker will find and link *every* object file whose name starts with *MYPROG*. (As I mentioned earlier, object files are unneeded after they have been linked. Your *BUILD* file should generally contain commands to delete them.)

Debugger

Sometimes a program doesn't do what you expected, but you don't know quite why. If you can't spot the problem by examining the assembler listing, you probably need the services of the *debugger*. The debugger is a handy utility that lets you run a program one instruction at a time, or keep running the program until you tell it to stop. Each time the debugger executes an instruction, it highlights the instruction it will execute next and shows the current values of the registers and the data on the stack.

The debugger can also display the contents of memory and *disassemble* programs; that is, it can display an assembled and linked program in its mnemonic form. Disassembling is convenient, for example, for examining programs stored in ROM.

Starting the Debugger

To start the debugger with the *APW* prompt on the screen, enter the command *debug*. When the debugger's initial screen appears, press Return to clear the copyright information at the bottom. Now you must load the program you want to debug. To do this, put the program's disk in the drive and enter a command of the form

```
load pathname
```

where *pathname* is the program's pathname.

Since you only have one program at this point, load it by entering **load beep**. When BEEP has been loaded, the screen will look similar to Figure 2-4.

Subdisplays on the Debugger Screen

The *Apple IIGS Programmer's Workshop* manual refers to the entire screen as the “master display,” and to areas on the screen as “subdisplays.” The two lines at the top of the screen are called the *register subdisplay*. They show the following:

- KEY is a keystroke modifier. The debugger normally interprets keystrokes as commands to itself (more about these commands shortly). Thus, if the program you are debugging involves any kind of keyboard input, you must tell the debugger that a keystroke is for your use by pressing another key (a “modifier”) with it.

```

KEY BRK DebugD K/PC B D S A X Y M Q L P nvmxdizc e
00 a d 1400 01/0CE9 00 2C00 2FFF 100A 0000 0000 00 9E 0 00 00000000 0

3011:D0 00/0000: 71 'q' 00/0000-00-00
3010:BF 00/0000: 71 'q' 00/0000-00-00
300F:7E 00/0000: 71 'q' 00/0000-00-00
300E:80 00/0000: 71 'q' 00/0000-00-00
300D:0B 00/0000: 71 'q' 00/0000-00-00
300C:A2 00/0000: 71 'q' 00/0000-00-00
300B:2C 00/0000: 71 'q' 00/0000-00-00
300A:B0 00/0000: 71 'q' 00/0000-00-00
3009:4A 00/0000: 71 'q' 00/0000-00-00
3008:60 00/0000: 71 'q'
3007:2D 00/0000: 71 'q' E1/0000.000F-T
3006:B0 00/0000: 71 'q' 00/0000.0000-?
3005:FD 00/0000: 71 'q' 00/0000.0000-?
3004:2B 00/0000: 71 'q' 00/0000.0000-?
3003:20 00/0000: 71 'q' 00/0000.0000-?
3002:06 00/0000: 71 'q' 00/0000.0000-?
3001:90 00/0000: 71 'q' 00/0000.0000-?
3000:02 00/0000: 71 'q' 00/0000.0000-?
2FFF:C9 00/0000: 71 'q' 00/0000.0000-?

: (Command line)

```

Figure 2-4

You can make the debugger recognize Shift, Control, Caps Lock, Option, OpenApple, a keypad key, or a repeating key as a keystroke modifier. Details are in the *Programmer's Workshop* manual.

- **BRK** is a breakpoint flag. A breakpoint is a location in a program at which the debugger is to stop executing and let you examine the registers or memory. I discuss breakpoints later.
- **DebugD** points to a direct page that the debugger uses internally. You can generally ignore it.
- The next seven entries are the 65816's registers: **K/PC** is program bank register and program counter, **B** (data bank register), **D** (direct page register), **S** (stack pointer), **A**, **X**, and **Y**. Here, **K/PC** points to the first instructions in your program, although that instruction is not yet displayed on the screen.
- **M** and **Q** are the so-called machine-state and quagmire registers. These registers aren't very useful for most applications.
- **L** is the language card bank.
- **P** is the processor status register. Its individual bits are shown at the end of the register subdisplay. Note that *m*, *x*, and *e* are 0; the debugger assumes you want full native mode.

The debugger provides commands that let you change the contents of any register, and I will discuss them shortly.

The leftmost column on the screen is the *stack subdisplay*. It shows the address and contents of the memory locations that precede and follow the location pointed to by the stack pointer. The entry in inverse video at the bottom indicates the current stack location.

The columns that extend to the end of the **K/PC** data comprise the *RAM subdisplay*. By entering **mem**, you can display the contents of up to 19 different locations in memory — one memory entry on each line. Once the cursor is in the RAM subdisplay, move to the line you want and type a hexadecimal address; the digits shift left as you type them. Then type one of three letters:

- **H** displays the 1-byte hex contents of that location and its corresponding ASCII character (e.g., FA 'z').
- **P** displays the contents of the location and the next location as a 4-digit number (e.g., E9E6).

- *L* displays the contents of the location and the next two locations as a 6-digit number (e.g., 008721).

To return to the command line, press **Esc**.

The rightmost column in Figure 2-4 contains two subdisplays: the *breakpoints subdisplay* at the top and the *memory protection subdisplay* at the bottom. I discuss the breakpoints subdisplay later, under “Breakpoints.” The memory protection subdisplay is only useful for advanced applications, but you might like to read about it in the *Programmer’s Workshop* manual.

The blank area at the right of the screen is reserved for the *disassembly subdisplay*, in which the debugger will display your program. Now it’s time to look at the available debugger commands.

Table 2-8 lists the most commonly used debugger commands. Note that they are divided into six groups, arranged in order of importance.

Single-Step and Trace Commands

These commands let you execute a program one instruction at a time (single step) or execute it until you tell the debugger to stop (trace). In either case, you can tell the debugger to start at the current K/PC address or at another address of your choice.

The debugger will accept an address in nearly any form, as long as you follow it with an S or T. Hence, 104EDS, 0104EDS, 1/04EDS, and 01/04EDS are all valid commands to enter step mode at location \$04ED of bank 1. If you omit the bank number (and enter, say, 04EDS), the debugger assumes you’re referring to the current program bank, so it obtains the bank number from the program bank register (K).

Once the debugger is in single-step or trace mode (*SINGLE STEP* or *TRACE* appears at the bottom of the screen), you can enter single-key “subcommands” to trace to the end of a subroutine, skip the next instruction (usually BRK), turn the sound on or off, suspend tracing, or change the trace speed.

The *T* subcommand, which changes the screen to text mode, is important for any program that displays graphics. Once the program starts running, it switches the screen from the debugger’s display to the application’s, and you lose sight of the instructions being traced. The *T* subcommand puts the screen back into the debugger, where you can once again follow the trace operation.

Table 2-8

Command	Description
Single-Step and Trace Commands	
S	Enter single-step mode at the current instruction. The current instruction. The current instruction is the one the 65816 will execute next, the one K/PC points to; the debugger highlights it on the screen. To execute that instruction, press the spacebar; to leave single-step mode, press Esc.
<i>address</i> S	Enter single-step mode at <i>address</i> .
T	Enter trace mode at the current instruction. The debugger begins executing immediately, and stops only when you press Esc or it encounters a breakpoint or BRK instruction.
<i>address</i> T	Enter trace mode at <i>address</i> .
The following are subcommands. They are only available when the debugger is in single-step or trace mode.	
Esc	Exit single-step or trace mode.
OpenApple	Stop tracing until the OpenApple key is released.
Spacebar	Execute highlighted instruction (in single-step mode).
R	Trace up to the next RTS, RTI, or RTL instruction. This lets you trace through one subroutine at a time.
T	Change the display to text mode.
Left-arrow	Change to the slow trace speed.
Right-arrow	Change to the fast trace speed (default).
Down-arrow	Skip the next instruction. This is convenient for skipping over a BRK, for example.
Q	Turn the sound off if it is on, or vice versa.
Editing Commands	
Control-E	Toggle insert/replace mode. In insert mode, typed characters are inserted between existing characters; in replace mode, typed characters replace existing characters.
Delete	Delete the character to the left of the cursor.
Control-F	Delete the character at the cursor position.
Esc	Delete the current line.
Control-Y	Delete to the end of the line.
Return	Execute the command. The cursor can be anywhere on the line.

Table 2-8 (cont.)

Command	Description
Register Commands	
e	Toggle the emulation mode flag.
m	Toggle the index register flag.
x	Toggle the memory/accumulator flag.
<i>register = value</i>	<p>Load the specified register with the specified value. Register names are:</p> <ul style="list-style-type: none"> KEY — Key modifier K — Program bank register PC — Program counter B — Data bank register D — Direct page register S — Stack pointer A — Accumulator X — Index register X Y — Index register Y M — Machine-state Q — Quagmire L — Language card bank P — Processor status register
Memory Commands	
<i>address:</i>	<p>Display 368 consecutive bytes in memory, starting at <i>address</i>. The debugger also shows the ASCII equivalent of each byte in either normal video (values between \$20 and \$7F) or inverse video (values between \$C0 and \$FF). ASCII values that can't be displayed appear as either a period (\$00 to \$1F) or an inverse period (\$80 to \$BF).</p>
<i>address:value(s)</i>	<p>Store the hexadecimal <i>value(s)</i> in memory, starting at <i>address</i>. For example, the following command stores \$A1 at location \$100 in bank 1:</p> <p>01/0100:A1</p> <p>You can store up to three bytes at a time. For example, the following command stores \$A1 at 01/0100, \$A2 at 01/0101, and \$A3 at 01/0102:</p> <p>01/0100:A1A2A3</p>
<i>address:'string'</i>	<p>Store the values corresponding to <i>string</i> in memory, starting at <i>address</i>. The high bit of each byte is 0, which produces normal video if you display the string.</p>
<i>address:"string"</i>	<p>Store the values corresponding to <i>string</i> in memory, starting at <i>address</i>. The high bit of each byte is 1, which produces inverse video if you display the string.</p>

Table 2-8 (cont.)

Command	Description
Memory Commands (cont.)	
<i>address:instruction</i>	Assemble the specified <i>instruction</i> and store its opcode and operand at <i>address</i> . The debugger will display the disassembled form of the new instruction when you enter single-step or trace mode.
Disassembly Commands	
<i>address</i> L	Disassemble 19 instructions, starting at <i>address</i> .
L	Disassemble the next 19 instructions, starting at the current K/PC address.
ASM	Clear the disassembly subdisplay. This only removes the disassembled instructions from the screen; it does not affect K/PC.
Conversion Commands	
<i>value</i> =	Convert <i>value</i> from hexadecimal to decimal. The <i>value</i> can range from 0 to FFFF.
+ <i>value</i> =	Convert <i>value</i> from decimal to hexadecimal. The <i>value</i> can range from 0 to 65535.
- <i>value</i> =	Convert <i>value</i> from decimal to hexadecimal. The <i>value</i> can range from 0 to -32768.

Editing Commands

The editing commands let you correct typing errors on the command line. Note that except for Esc, these are the same commands you use in the editor.

Register Commands

You can change any register by entering a command of the form

register = *value*

where *register* is the abbreviation for the register (it must be uppercase) and *value* is a hexadecimal number. For example, X = 10 loads \$10 (decimal 16) into the X register. You may want to change a register at some point in a program to see what would happen under some alternate set of circumstances. You may also want to change an index register to terminate a long loop operation that you don't care about.

You can also reverse, or "toggle," the setting of the status register's

emulation, memory/accumulator, or index register bit by entering *e*, *m*, or *x*, respectively. You probably won't do this very often, however; the program should regulate these bits.

Memory Commands

These commands let you display and, optionally, alter the contents of memory. The *address:* command makes the debugger display a screenful of bytes in memory (368 bytes, to be exact), starting at the specified address. It also shows the ASCII equivalent of each byte, where applicable, and shows non-character bytes as either a period (.) or inverse period. To leave the memory display and return to the regular debugger screen, press Esc.

Of course, the *mem* command also displays memory, on the RAM sub-display, but it can only show 3 bytes on a line — or 1 byte, if you want the ASCII representation. The *address:* command provides much more information, but it doesn't let you see the registers or your program at the same time. Alas, nothing comes for free.

You can also change data in memory, by entering a hexadecimal value or string after *address:*. Finally, if you enter an instruction after *address:*, the debugger will assemble it and store its opcode and operand starting at that location. For example,

```
0834:LDY #0010
```

stores \$A0 (opcode for LDY immediate), \$10, and \$00 at locations \$834, \$835, and \$836 of the current program bank. The operand 0010 tells the debugger that the 65816 is running in native mode with 16-bit index registers (status flag X=0). The debugger isn't smart enough to know that the shorter form, LDY #10, means the same thing if Y is 16 bits long, because it doesn't look at the X flag. It simply assembles what you give it.

Disassembly Commands

Sometimes you may want to look at instructions in other parts of memory, say, a subroutine in ROM or instructions in another part of your own program. To do this, you can make the debugger “disassemble” memory starting at a specified address by entering *addressL*. Figure 2-5 shows a typical disassembly that was produced with the command 01/1200L. The debugger always disassembles 19 instructions, but only the first eight are shown here.

Note that for the program counter relative mode and relative long


```

12/1000: AD 15 18      LDA 1815
12/1003: 9D 50 10      STA 1050, X
12/1006: 9F 20 30 05    STA 050320
12/100A: A9 77 66      LDA #6677
12/100D: 82 20 10      BRL 2030 (+1020)
12/1010: 80 20         BRA 1032 (+20)
12/1012: F4 12 34      PEA 3412
12/1015: 62 10 10      PER 2028 (+1010)
...
..  11 more instructions
..

```

Figure 2-5

modes, as used by the BRL, BRA, and PER instructions, the debugger lists the effective address followed in parentheses by the relative displacement. You probably won't care about the displacements, but they're there if you ever need them.

Conversion Commands

Finally, the debugger can perform on-screen conversions of hex-to-decimal or vice versa. You might use one of these commands to determine the decimal form of a hexadecimal operand that the debugger has displayed. Once you give a conversion command, it remains on the command line until you press Esc.

Breakpoints

As mentioned earlier, there are two ways to execute a program using the debugger. You can single step through it one instruction at a time, or you can trace through it, executing instructions until the processor encounters a BRK instruction or you press Esc. The debugger also lets you set *breakpoints* in a program.

A breakpoint is a location at which the debugger is to stop tracing so you can examine the contents of registers and memory. Breakpoints can be valuable for locating an elusive "bug" that's making your program produce incorrect answers or crash. They put *you* (rather than the microprocessor) in control of the program.

How you use breakpoints will depend on your application. You might set one at the instruction that follows a loop, to see what the loop produced

before continuing. You could also set a breakpoint to determine whether a particular instruction is ever executed; if tracing never stops there, the instruction has not been executed.

The debugger lets you set up to 17 breakpoints. The breakpoints subdisplay (above the memory protection subdisplay) has space for nine entries, but you can enter more by using some of the memory protection subdisplay's lines.

Entries in the breakpoints subdisplay are initially set to *00/0000-00-00*, where the first item is (as you may have guessed) the breakpoint address. The second item is the "trigger value"; the number of times the debugger is to execute the breakpoint instruction before stopping. The third item is a repetition counter. During tracing, it displays a running total of the number of times the instruction has been executed.

To set a breakpoint, enter **bp** on the command line. This moves the cursor to the hyphen that separates the address and trigger value in the first entry. To enter a breakpoint there, type its address, then press the right-arrow key and type the trigger value. If that's the only breakpoint you want, press Esc to return to the command line; otherwise, press the down-arrow key to reach the next line where you want to enter a breakpoint. If you make a mistake entering a breakpoint or decide you don't want it, move to its line and either reenter it or press Delete to clear it.

You can also enter three breakpoint commands from the command line: *clr* clears all breakpoints, *in* makes the debugger insert BRK instructions at the breakpoint locations (the register subdisplay's BRK entry shows "i"), and *out* removes the BRKs that *in* inserted (the BRK entry shows "o").

Leaving the Debugger

To leave the debugger and restore the # prompt, press *Q* for Quit.

CHAPTER 3

65816 Addressing Modes

The 65816 provides 24 different ways to obtain the data on which your program is to operate. Table 3-1 lists these addressing modes in the order I describe them in this chapter. It also gives the assembler format for each mode (which is how the 816 differentiates them) and shows, with shading, which modes are new with the 816.

In this table, *Loc* and *LongLoc* represent 16- or 24-bit addresses in a program bank or data bank (which one depends on the instruction), while *DLoc* and *DLongLoc* represent 16- or 24-bit addresses in the direct page. These memory address operands are generally labels, and the assembler converts them to the numeric addresses they represent.

For example, suppose your program contains the instruction `JMP NEWLOC` (i.e., jump or GOTO the instruction labeled NEWLOC), where NEWLOC is at location \$1200 in the program bank. When you assemble the program, the assembler will translate NEWLOC into the numeric address \$1200.

The terms *direct*, *indirect*, and *indexed* need a general introduction. Direct means that the address of the data is in the direct page (called the zero page in 6502 literature). Indirect means that the address of the data is in the specified location. Indexed means that the address of the data is obtained by adding the contents of an index register to the specified address.

Table 3-1

Mode	Operand Format
Immediate	#Num
Accumulator	A
Implied	<i>blank</i> (no operand)
Absolute	Loc
Absolute Long	LongLoc
Absolute Indirect	(Loc)
Absolute Indexed with X	Loc,X
Absolute Indexed with Y	Loc,Y
Absolute Long Indexed with X	LongLoc,X
Absolute Indexed Indirect	(Loc,X)
Direct (Zero Page)	DLoc
Direct Indirect	(DLoc)
Direct Indirect Long	[DLongLoc]
Direct Indexed with X	DLoc,X
Direct Indexed with Y	DLoc,Y
Direct Indirect Indexed	(DLoc),Y
Direct Indirect Indexed Long	[DLongLoc],Y
Direct Indexed Indirect	(DLoc,X)
Program Counter Relative	Disp8
Program Counter Relative Long	Disp16
Stack	(Various operands)
Stack Relative	Disp8,S
Stack Relative Indirect Indexed	(Disp8,S),Y
Block Move	srcbk, destbk

Notes: (1) Shaded modes are new with the 65816; they are not available on the 6502.

(2) Symbols have the following meanings:

Num = 8- or 16-bit constant

Loc = 16-bit address

LongLoc = 24-bit address

DLoc = 16-bit address in the direct page

DLongLoc = 24-bit address in the direct page

Disp8 = 8-bit signed relative displacement (distance forward or backward)

Disp16 = 16-bit signed relative displacement

srcbk = Source bank number

destbk = Destination bank number

Immediate

The immediate addressing mode lets you specify a *constant* as the operand. Here, you must precede the constant with a # symbol. For example, the instruction

```
LDA    #$4A
```

loads the hexadecimal value 4A (decimal 74) into the accumulator (the A register).

Sometimes you will want to work with the *address* of a memory location, rather than its contents. (Address operations are common in the IIGS Toolbox.) Memory addresses are 3 bytes long — a 1-byte bank number and a 2-byte offset — and are stored in offset/bank number order in memory. Thus, to read the offset into a register, you would do a load immediate of the location's label (e.g., LDA #ThisLabel); to read the bank number, you would do a load immediate of the label *plus* 2 (e.g., LDA #ThisLabel + 2).

Accumulator

In the accumulator mode, the operand is in the accumulator. For example, this instruction increments the accumulator (adds 1 to it):

```
INC A
```

Instructions that use the accumulator addressing mode are only 1 byte long, because the opcode provides all the information the processor needs.

Implied

About half of the 65816's instructions perform simple tasks such as setting or clearing a bit in the processor status register, incrementing or decrementing a register, or copying the contents of one register into another. These instructions need no operand because the operand is “implied” in the opcode. Some examples are:

```
CLC      ;Clear Carry Flag
DEY      ;Decrement the Y Register
INX      ;Increment the X Register
TAY      ;Transfer Accumulator to Y Register
```

All implied addressing instructions are 1 byte long.

Absolute and Absolute Long

Absolute addressing allows you to access any of the 64K locations in a bank of memory. Absolute addressing instructions are 3 bytes long: the opcode,

followed by a 16-bit address. When you're accessing data, the 65816 uses the data bank register (DBR) to select the bank. For example, if the location `DATALOC` is in the active data bank,

```
LDA DATALOC
```

reads its contents into the accumulator.

The instructions `Jump (JMP)` and `Jump to Subroutine (JSR)` — the assembly language counterparts of `GOTO` and `GOSUB` in BASIC — also use absolute addressing. However, because they are used to move within a program, the 65816 employs the program bank register (PBR), rather than the data bank register, to select the bank. For example,

```
JMP THERE
```

makes the 816 continue at the instruction that is labeled `THERE`, by putting its address in the program counter.

You may be wondering what happens if the target location or instruction is in a different bank than the one to which the bank register (DBR or PBR) is pointing. When that happens, the assembler encodes the instruction using the *absolute long* mode. In absolute long mode, the instruction contains a 24-bit address that points to your target location. Here, the high-order 8 bits of the address specify the bank, while the remaining 16 bits specify the location within that bank. For example, if `DATALOC` is not in the active data bank, the assembler assembles

```
LDA DATALOC
```

using absolute long addressing.

The key point is that you needn't be concerned about *where* your data is located. The assembler will determine its location and apply the correct mode, absolute or absolute long, automatically.

In general, absolute operands are labels, but you may (for some reason) want to specify the numeric form of an address. To do this, simply enter the number directly. For example, `LDA $100` reads the contents of location \$100 in the active data bank. However, to use a numeric operand for absolute long addressing, you must precede it with a `<` symbol. For example, `LDA <$100` reads the contents of location \$100 in bank 0.

Absolute Indirect

Absolute indirect is really two addressing modes, one used only by the JMP instruction and the other used only by the JML instruction. (As mentioned in the preceding section, JMP is the assembly language version of BASIC's GOTO. JML, short for JumpLong, is the same, except it can transfer to any bank; JMP is limited to the current bank.) The term *indirect* here indicates that the operand *contains* the target address; the operand is not itself the target address, as it is for a JMP absolute instruction.

In the case of JMP, the indirectly addressed location contains a 2-byte address in the 65816's standard order: with the high byte following the low byte. The 65816 combines this address with the 8-bit program bank register (PBR) to produce the 24-bit destination address. For example,

```
JMP (TADD)
```

makes the 65816 transfer to the location in the active program bank whose address is contained in the 16-bit location TADD. Figure 3-1 shows how this instruction operates if TADD contains \$014C.

In the case of JML, the indirectly addressed location contains all 3 bytes of the destination address, with the low byte first. To make the transfer, the 65816 loads the low and middle bytes into the program counter (PC) and the high byte into the program bank register (PBR).

You may be thinking, "Why all this rigmarole? If your destination is some specific address, why not simply use regular absolute long addressing to load it into the program counter and PBR?" The answer is that absolute

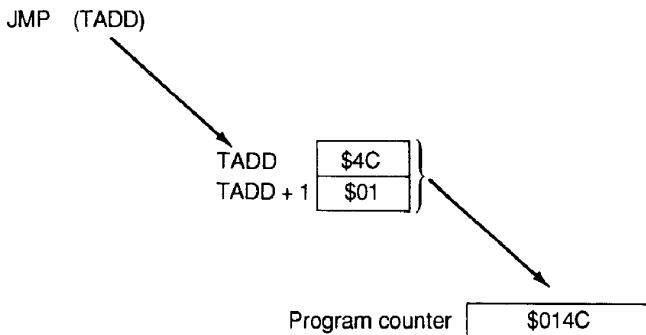


Figure 3-1

indirect addressing allows you to work with *variable* destination addresses. For example, in a program where a user chooses from options in a menu, you can use the option number to look up the address of that option's sub-program in a table, store the address in memory, then jump to it using absolute indirect addressing. Absolute indirect addressing also allows the effective address to be in read/write memory (RAM) even when the program is in ROM or PROM.

The absolute indirect mode has one restriction that limits its use: *the location that holds the indirect address must be in bank 0!* That being the case, you may want to make indirect jumps using the absolute indexed indirect mode (discussed shortly), which doesn't have the bank 0 restriction.

Absolute Indexed with X or Y

In absolute indexed addressing, the 65816 computes the effective address by adding the contents of an index register (X or Y) to the absolute address in the instruction. That is,

$$\text{Effective address} = \text{Absolute address} + X$$

or

$$\text{Effective address} = \text{Absolute address} + Y$$

All absolute indexed instructions are 3 bytes long. Absolute indexed operands are formed by attaching a ,X or ,Y to the address or address label.

Absolute indexed addressing is particularly useful for accessing tables. Here, you would enter the table's name as the address operand and use an index register to select the item you want. For example, if DTABLE is a table of 16-bit words and X contains 6, this instruction

```
LDA DTABLE,X
```

loads the third word of DTABLE into the accumulator. Figure 3-2 shows how this instruction operates.

Note that it is necessary to put 6 into X because you are referring to a table of words (rather than bytes), and words are 2 bytes long. Using 3 here would cause the 816 to load the third and fourth bytes of DTABLE into the accumulator, and that's not what you intended.

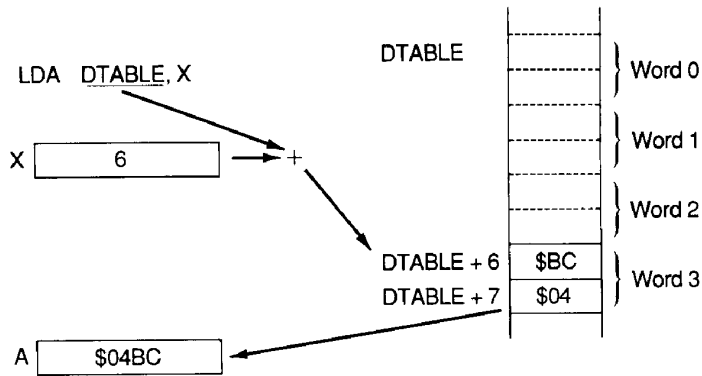


Figure 3-2

Absolute Long Indexed with X

The absolute indexed modes just described always obtain their operands from the active data bank — that is, from the bank to which the data bank register (DBR) is pointing. If you give an absolute indexed with X instruction that refers to a location in some other bank, the assembler assembles it using the absolute *long* indexed with X mode. In that case, the instruction is 4 bytes long, rather than 3, because it contains a 3-byte (24-bit) address. Note that there is no comparable mode for the Y register.

If your operand is a number rather than a label, you must tell the assembler which mode to use by preceding the absolute long indexed operand with a `>` symbol. For example, `LDA $100,X` refers to location \$100 in the active data bank, while `LDA >$100,X` refers to location \$100 in bank 0.

Absolute Indexed Indirect

This is a combination of two modes that were discussed earlier: absolute indirect and absolute indexed with X. Recall that with absolute indirect addressing, the operand is the address of the location that *contains* the effective address. Recall also that absolute indexed with X involves adding a displacement in the X register to a base address in the instruction to produce the effective address.

With absolute indexed indirect addressing, the 65816 adds the contents

of the X register to the absolute address in the instruction to obtain the indirect address. That is,

$$\text{Indirect address} = \text{Absolute address} + X$$

then

$$\text{Effective address} = (\text{Indirect address})$$

where the parentheses mean “contents of.” Absolute indexed indirect operands have the general form (Loc,X).

To see how this mode is used, suppose your program displays a menu and prompts the user to type a number from 0 to 4 to select an option. Suppose also that it has a table called JTABLE that contains five 16-bit addresses, one for each option. To begin, the program reads the number into the X register and doubles it (because you’re accessing words, not bytes). Then, to start the routine for the selected option, it executes:

```
JMP (JTABLE,X)
```

where JTABLE is assumed to be in the current program bank (*not* the data bank). Figure 3-3 shows how this instruction operates if X contains 4.

Direct

Within bank 0 of memory, there is a certain page (i.e., a 256-byte block of locations) that the 65816 can access faster than any other page. If you think

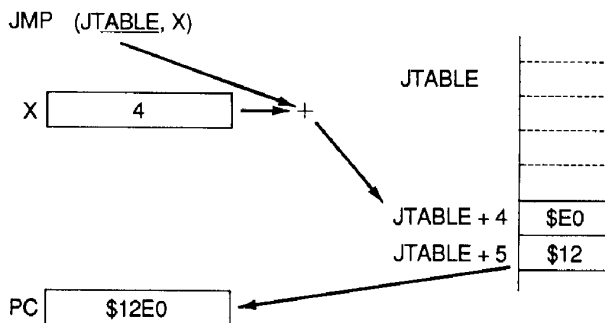


Figure 3-3

of the computer's memory as a group of boxes in a post office — one for each mail carrier in a city — consider this “direct” page (or *zero page*, in 6502 terminology) as the box that is closest to the person sorting mail. Because of this direct box's proximity, the sorter can toss items into it immediately, without even thinking. However, whenever the sorter encounters an item that belongs in any other box, he or she must first search for the box, and that takes more time.

The 65816 acts exactly like the mail sorter in my hypothetical post office; it can transfer data between the direct page — the page in bank 0 at which the direct register (D) is pointing — and a register faster than it can transfer to or from any other page. Thus, when you give an instruction that refers to a location in the direct page, the assembler says, “Oh, that's in the direct page. The 816 can do that operation faster than normal.” It then assembles the instruction using *direct* addressing.

Direct addressing is, then, a form of absolute addressing in which the 65816 accesses only a specific 256-byte page within bank 0. Direct addressed instructions are one byte shorter and one clock cycle faster than absolute addressed instructions. To construct the operand address, the 65816 adds the 8-bit offset in the instruction to the 16-bit contents of the direct register (D). Again, the assembler will use direct addressing rather than absolute addressing automatically, where applicable; you needn't tell it which mode to use.

Except for JMP (Jump) and JSR (Jump to Subroutine), all instructions that can use absolute addressing can also use direct addressing. However, considering the savings in storage space and execution time with direct addressing, you should, whenever possible, use the direct page to hold frequently accessed data. The direct page is also useful for storing temporary data values.

Direct Indirect and Direct Indirect Long

With direct indirect addressing, the processor adds an 8-bit offset contained in the instruction to the 24-bit contents of the direct register to produce an address within the direct page. It then combines the 16-bit contents of the addressed location with the 8-bit data bank register (DBR) to produce the 24-bit effective address of the data. The operands for direct indirect instructions have the general form (DLOC). For example, if DLOC is in the direct page,

```
LDA (DLOC)
```

uses the contents of DLOC as an address in the active data bank from which it obtains the data to be loaded into the accumulator.

The direct indirect long mode is the same, except the processor obtains a 24-bit (3-byte) address from the indirect location. Direct indirect long operands have the general form [DLongLoc]. For example,

```
LDA [DLOC]
```

Direct Indexed with X or Y

In direct indexed addressing, the 65816 computes the effective address by adding the contents of the direct register (D) and an index register (X or Y) to the 8-bit offset in the instruction. That is,

$$\text{Effective address} = \text{Offset} + D + X$$

or

$$\text{Effective address} = \text{Offset} + D + Y$$

All direct indexed instructions are 2 bytes long, and their operands are formed by attaching a ,X or ,Y to the offset.

Direct indexed addressing is particularly useful for accessing tables. Here, you would enter the table's name as the offset operand and use an index register to select the item you want. For example, if DTABLE is a table of 16-bit words and X contains 6, this instruction

```
LDA DTABLE,X
```

loads the third word of DTABLE into the accumulator.

Direct Indirect Indexed and Direct Indirect Indexed Long

Direct indirect indexed addressing is similar to direct indirect addressing, except that after obtaining the indirect address, the 65816 adds the contents of the Y register to it to form the 24-bit effective address. That is:

$$\text{Effective address} = (\text{DLOC}) + Y$$

This is useful if the indirectly addressed location points to a table. Then, adding the Y register produces the address of an element in the table. Direct indirect indexed operands have the general form (DLoc),Y.

Direct indirect indexed long is the same, but the 816 extracts a 24-bit address, rather than a 16-bit address, from the indirectly addressed location. Direct indirect indexed long operands have the general form {DLongLoc},Y.

Direct Indexed Indirect

The direct indexed indirect mode is somewhat like the similar-sounding direct indirect indexed mode, except that the 65816 uses the X register to calculate the location in the direct page from which it is to obtain the address of the data. That is,

$$\text{Indirect address} = D + \text{offset} + X$$

then

$$\text{Effective address} = (\text{Indirect address})$$

Direct indexed indirect operands have the general form (DLoc,X).

This mode is useful for selecting an address from a table of addresses in the direct page. If ADDR_TBL is such a table,

```
LDA  (ADDR_TBL,X)
```

extracts the address at offset X and uses it to obtain the operand that it loads into the accumulator.

Program Counter Relative and Program Counter Relative Long

Program counter relative addressing is just what its name implies: the effective address is some offset from the current value of the program counter (PC). Since the 65816 updates the PC before it executes an instruction, the PC will be pointing to the *next* instruction when the offset is applied. A positive offset produces a higher-numbered address; a negative offset produces a lower-numbered address.

Program counter relative addressing is only used by the 65816's *branch* instructions. These instructions make the 816 transfer forward or backward if a specified condition is met (e.g., if a preceding add operation produced a zero result). Otherwise, if the condition is not met, the 816 proceeds to the next instruction. For example, the following BCC (Branch on Carry Clear) instruction will make the 816 transfer to the instruction at NEXT if the carry flag is 0, or to NOTTEXT if carry is 1:

```

                                BCC    NEXT    ; Is carry = 0?
NOTNEXT    LDA    #$3A    ; No. Continue here.
                                .
                                .
                                .
NEXT       LDA    #4      ; Yes. Transfer here.
```

Branch instructions are two bytes long, where the second byte contains the signed offset. With an offset of 8 bits, regular branch instructions can only transfer 127 bytes forward or 128 bytes backward — about half a page in either direction. However, there is a Branch Long (BRL) instruction that can transfer halfway up or down the program bank. BRL uses *program counter relative long* addressing, which adds a 16-bit offset to the PC.

Stack

This is actually several different modes that are used by various instructions that access the stack, either to “push” data onto it or “pull” data off it.

Stack Relative and Stack Relative Indirect Indexed

These are two modes that you can use to access data on the stack. These modes are rarely used, however; most people only put data on the stack or retrieve data from it. Still, it's nice to know that the 65816 provides some stack-accessing capability if you ever need it.

With stack relative addressing, the 816 adds an 8-bit signed offset in the instruction to the 16-bit contents of the stack pointer (S) to produce an effective address in bank 0. Stack relative operands have the general form `Disp8,S`.

Stack relative indirect indexed addressing is similar, except that the sum of the offset and the stack pointer is an *indirect* address. The 65816 adds

the value in the Y register to the contents of the indirectly addressed location to produce the effective address. Stack relative indirect indexed operands have the general form (Disp8,S),Y.

Block Move

This mode is only used by the Block Move Negative (MVN) and Block Move Position (MVP) instructions. These instructions copy a specified number of bytes from a “source” bank to a “destination” bank, starting at either the beginning of the source block (MVN) or the end of it (MVP).

The two distinct block move instructions are designed to keep you from overwriting data. For example, if you move a block forward one byte position in memory (to the next higher numbered address), and start at the beginning of the block, the first byte you move will overwrite the second byte. Instead, you must start moving from the end of the block, using MVP.

Similarly, if you move a block downward one byte position (to the next lower numbered address), and start at the end of the block, the first byte you move will overwrite the preceding byte. This time you must start moving from the beginning of the block, using MVN.

The operands for the block move instructions have the general form *srcbk,destbk*, where *srcbk* and *destbk* are the numbers of the source and destination banks, in hexadecimal. For each instruction, the X register contains the offset of its destination, and the A register specifies the number of bytes to be moved *minus one*. Hence, if X and Y contain 4 and A contains 99, the instruction

```
MVP 5,6
```

tells the 65816 to copy 100 bytes from bank 5 to bank 6, starting with the fifth byte (byte 4).

Addressing Mode Summary

As you now know, the addressing modes that access memory have many variations, and they can be somewhat confusing. To help sort things out, I have included Table 3-2. This table lists the name and operand format for each mode, but in addition it shows — in equation form — how the 65816 calculates the actual address of an operand; that is, its *effective address*.

Table 3-2

Mode	Operand Format	Effective Address =
Absolute	Loc	Loc
Absolute Long	LongLoc	LongLoc
Absolute Indirect	(Loc)	(Loc)
Absolute Indexed with X	Loc,X	Loc + X
Absolute Indexed with Y	Loc,Y	Loc + Y
Absolute Long Indexed with X	LongLoc,X	LongLoc + X
Absolute Indexed Indirect	(Loc,X)	(Loc + X)
Direct (Zero Page)	DLoc	DLoc
Direct Indirect	(DLoc)	(DLoc)
Direct Indirect Long	[DLongLoc]	(DLongLoc)
Direct Indexed with X	DLoc,X	DLoc + X
Direct Indexed with Y	DLoc,Y	DLoc + Y
Direct Indirect Indexed	(DLoc),Y	(DLoc) + Y
Direct Indirect Indexed Long	[DLongLoc],Y	(DLongLoc) + Y
Direct Indexed Indirect	(DLoc, X)	(DLoc + X)
Program Counter Relative	Disp8 or Loc	PC + Disp ⁸
Program Counter Relative Long	Disp16 or Loc	PC + Disp16
Stack Relative	Disp8,S	S + Disp8
Stack Relative Indirect Indexed	(Disp8,S),Y	(S + Disp8) + Y

Notes: (1) Shaded modes are new with the 65816; they are not available on the 6502.

(2) Symbols have the following meanings:

Loc = 16-bit address.

LongLoc = 24-bit address.

DLoc = 16-bit address in the direct page.

DLongLoc = 24-bit address in the direct page.

Disp8 = 8-bit signed relative displacement (distance forward or backward).

Disp16 = 16-bit signed relative displacement.

For example, in an 8-bit mode, an immediate instruction such as LDA #10 will execute in two cycles and occupy two bytes in memory, versus three cycles and three bytes in full native mode. Appendix C gives time and storage data for each instruction.

Note that the table also correlates each addressing mode with the microprocessor on which it was introduced. This is intended to benefit readers who have done some assembly language work on earlier models of the Apple II, and those who are writing programs which are to be 6502 and 65C02 compatible.

Read-Modify-Write Instructions

Four of the modes in Table 3-3 have increased cycle times when they are being used with a Read-Modify-Write instruction. As the name implies,

Table 3-3

Addressing mode	Example	Cycles	Bytes	Introduced in		
				6502	65C02	65186
Accumulator	INC A	2	1	x		
Immediate	LDA #30	3	3	x		
Implied	INX	2	1	x		
Absolute					x	
<i>Read-Modify-Write ins.</i>	INC Loc	8	3			
<i>Other instructions</i>	LDA Loc	5	3			
Absolute Long	LDA LongLoc	6	4			x
Absolute indexed with X				x		
<i>Read-Modify-Write ins.</i>	INC Loc,X	8 ¹	3			
<i>Other instructions</i>	LDA Loc,X	5 ¹	3			
Absolute long indexed with X	LDA LongLoc,X	6	4			x
Absolute indexed with Y	LDX Loc,Y	5 ¹	3	x		
Absolute indirect	JMP (Loc)	5	3	x		
Absolute indexed indirect	JMP (Loc,X)	6	3		x	
Direct				x		
<i>Read-Modify-Write ins.</i>	INC DLoc	7 ³	2			
<i>Other instructions</i>	LDA DLoc	4 ³	2			
Direct indexed with X				x		
<i>Read-Modify-Write ins.</i>	INC DLoc,X	8 ³	2			
<i>Other instructions</i>	LDA DLoc,X	5 ³	2			
Direct indexed with Y	LDX DLoc,Y	5 ³	2	x		
Direct indirect	LDA (DLoc)	6 ³	2		x	
Direct indirect long	LDA [DLongLoc]	7 ³	2			x
Direct indirect indexed	LDA (DLoc),Y	6 ^{1,3}	2	x		
Direct indirect indexed long	LDA [DLongLoc],Y	7 ³	2			x
Direct indexed indirect	LDA (DLoc,X)	7 ³	2	x		
Relative	BEQ Label	2 ²	2	x		
Relative long	BRL Label	3 ²	3			x
Stack	PHA	3-8	1-4	x		
Stack relative	LDA 3,S	5	2			x
Stack rel. indirect indexed	LDA (4,S),Y	8	2			x
Block move	MVP Source, Dest	7	3			x

¹Add 1 cycle if adding index crosses a page boundary.

²Add 1 cycle if branch is taken.

³If direct register low (DL) does not equal zero, add 1 cycle.

Read-Modify-Write instructions operate by reading the contents of a memory location, changing it in some way, then returning the result to memory.

As the table shows, the INC (Increment) instruction is of the Read-

Modify-Write variety. *INC* adds one to the contents of an operand. If the operand is a memory location, *INC* must read the location's contents, add one to it, then write the result back to the original location. These extra tasks take time, and it's reflected in the higher cycle count. Among other Read-Modify-Write instructions are *DEX* (Decrement) and the shift and rotate instructions, which displace the contents of an operand to the left or right. These instructions, along with the rest of the 65816's instructions, will be covered in the next chapter.

Final Thoughts on Addressing Modes

If you are an old hand at Apple II assembly language, you probably breezed through the new 65816 modes. After all, except for the two stack relative modes (which hardly anyone would use) and the block move mode (which only applies to two instructions), the new modes simply provide for "long" operations — those involving inter-bank operations.

To assembly language newcomers, the addressing modes are a different story entirely. There are a lot of them, they often look alike (e.g., *LDA Loc,X* and *LDA Loc,Y*), and their names are sometimes baffling (e.g., direct indirect indexed long). In all, if you're just beginning in assembly language, your head may be spinning as you read this. That's natural, and you shouldn't worry about it.

In reality, you will probably use only these eight modes frequently:

- Accumulator
- Immediate
- Implied
- Absolute
- Absolute indexed with X
- Absolute indexed indirect
- Relative
- Stack

What's more, you will generally use a mode without knowing it. That's because in writing a program, you select the instructions that do what you want. Good programmers don't think in terms of choosing an addressing mode, but rather in terms of (A) which instruction and (B) which form of that instruction does what they want.

For example, to add one to the contents of the accumulator, or "increment" it, a programmer would use an *INC A* instruction, to which the

assembler applies the accumulator mode. Similarly, incrementing a memory location (say, *Count*) involves using an *INC Count* instruction, to which the assembler applies the absolute mode. Did the programmer consciously chose a mode? No. He or she simply selected an instruction, and left it up to the assembler to translate that instruction using the correct mode.

Just as my hypothetical programmer thinks in terms of instructions, so should you. Concentrate on finding an instruction that does the job, and let the assembler select the mode.

CHAPTER 4

65816 Instruction Set

In earlier chapters you became acquainted with LDA and some other simple instructions. In this chapter, I'll describe the 65816's entire instruction set. The 816 uses the same instructions as the earlier 6502 and 65C02, but it also has a few new ones. For the benefit of readers who have programmed on an Apple //e or some other 6502-based computer, I will note the differences.

In some books authors cover the instructions individually, discussing them in alphabetical order. This is suitable for reference manuals, but it tends to leave readers bored and bewildered after the fifth or sixth instruction. In this book, I group instructions by function, describing similar instructions together. For example, I group add instructions with subtract instructions, shifts with rotates, and so on. This should help you understand the instruction set and appreciate how individual instructions relate to each other, so you don't learn them as just a lot of disjointed entities.

Later, after running a few programs, you will only need to refer to this chapter occasionally, to look up details of specific instructions. Generally, you should be able to resolve most questions by referring to Appendix C, where the instructions are summarized alphabetically.

Instruction Types

As mentioned earlier, the 65816 has 91 different types of instructions. Table 4-1 shows their assembler mnemonics and tells what each stands for.

Alternate Mnemonics

The assembler also provides alternate mnemonics for certain instructions, as listed in Table 4-2. The most useful alternates are BLT (Branch if Less Than) and BGE (Branch if Greater Than or Equal) — branch instructions that test the result of a preceding compare operation. Certainly, people more often want to test whether an operand is “less than” or “greater than or equal to” another operand than whether the carry flag is 0 (clear) or 1 (set).

Table 4-1

Mnemonic	Meaning
ADC	Add to Accumulator with Carry
AND	AND Memory with Accumulator
ASL	Arithmetic Shift Left
BCC	Branch on Carry Clear ($C = 0$)
BCS	Branch on Carry Set ($C = 1$)
BEQ	Branch if Equal ($Z = 1$)
BIT	Bit Test
BMI	Branch if Minus ($N = 1$)
BNE	Branch if Not Equal ($Z = 0$)
BPL	Branch if Plus ($N = 0$)
BRA	Branch Always ¹
BRK	Force Break
BRL	Branch Always Long ²
BVC	Branch on Overflow Clear ($V = 0$)
BVS	Branch on Overflow Set ($V = 1$)
CLC	Clear Carry Flag
CLD	Clear Decimal Mode
CLI	Clear Interrupt Disable Bit
CLV	Clear Overflow Flag
CMP	Compare Memory and Accumulator
COP	Coprocessor ²
CPX	Compare Memory and X Register
CPY	Compare Memory and Y Register

Table 4-1 (cont.)

Mnemonic	Meaning
EOR	Exclusive-OR Memory with Accumulator
INC	Increment Memory or Accumulator
INX	Increment X Register
INY	Increment Y Register
JML	Jump Long ²
JMP	Jump
JSL	Jump to Subroutine Long*UP*2
JSR	Jump to Subroutine
LDA	Load Accumulator
LDX	Load X Register
LDY	Load Y Register
LSR	Logical Shift Right
MVN	Block Move Negative (to a lower-numbered address) ²
MVP	Block Move Positive (to a higher-numbered address) ²
NOP	No Operation
ORA	OR Memory with Accumulator
PEA	Push Effective Address onto Stack ²
PEI	Push Effective Indirect Address onto Stack ²
PER	Push Effective Program Counter Relative Address onto Stack ²
PHA	Push Accumulator onto Stack
PHB	Push Data Bank Register onto Stack ²
PHD	Push Direct Register onto Stack ²
PHK	Push Program Bank Register onto Stack ²
PHP	Push Processor Status Register onto Stack
PHX	Push X Register onto Stack ¹
PHY	Push Y Register onto Stack ¹
PLA	Pull Accumulator from Stack
PLB	Pull Data Bank Register from Stack ²
PLD	Pull Direct Register from Stack ²
PLP	Pull Processor Status Register from Stack
PLX	Pull X Register from Stack ¹
PLY	Pull Y Register from Stack ¹
REP	Reset Processor Status Bits ²
ROL	Rotate Left
ROR	Rotate Right
RTI	Return from Interrupt
RTL	Return from Subroutine Long ²
RTS	Return from Subroutine
SBC	Subtract Memory from Accumulator with Borrow
SEC	Set Carry Flag
SED	Set Decimal Mode
SEI	Set Interrupt Disable Bit
SEP	Set Processor Status Bits ²

Table 4-1 (cont.)

Mnemonic	Meaning
STA	Store Accumulator
STP	Stop the Clock ²
STX	Store X Register
STY	Store Y Register
STZ	Store Zero ¹
TAX	Transfer Accumulator to X Register
TAY	Transfer Accumulator to Y Register
TCD	Transfer C Accumulator to Direct Register ²
TCS	Transfer C Accumulator to Stack Pointer ²
TDC	Transfer Direct Register to C Accumulator ²
TRB	Test and Reset Bit ¹
TSB	Test and Set Bit ¹
TSC	Transfer Stack Pointer to C Accumulator ²
TSX	Transfer Stack Pointer to X Register
TXA	Transfer X Register to Accumulator
TXS	Transfer X Register to Stack Pointer
TXY	Transfer X Register to Y Register
TYA	Transfer Y Register to Accumulator
TYX	Transfer Y Register to X Register
WAI	Wait for Interrupt ²
WDM	Reserved for Future Use (No operation) ²
XBA	Exchange B and A Bytes of Accumulator ²
XCE	Exchange Carry and Emulation Bits (Switch modes) ²

¹Available only on the 65C02 and 65816, not on the 6502.²Available only on the 65816, not on the 6502 nor 65C02.

Table 4-2

Standard	Alias	Meaning
BCC	BLT	Branch if Less Than
BCS	BGE	Branch if Greater Than or Equal
CMP	CMA	Compare Memory and Accumulator
DEC A	DEA	Decrement Accumulator
INC A	INA	Increment Accumulator
TCD	TAD	Accumulator to Direct Register
TCS	TAS	Transfer C Accumulator to Stack Pointer
TDC	TDA	Transfer Direct Register to C Accumulator
TSC	TSA	Transfer Stack Pointer to C Accumulator
XBA	SWA	Swap Accumulator Bytes

Functional Groups

The instructions are divided into ten functional groups:

1. Data transfer instructions move information between registers, memory locations, and I/O devices.
2. Arithmetic instructions perform add and subtract operations on binary or binary-coded decimal (BCD) numbers.
3. Control transfer instructions can change the sequence in which a program executes; they include jumps and branches.
4. Subroutine instructions perform transfers to and from subroutines.
5. Stack instructions transfer data between registers and the stack, a data structure in memory.
6. Bit manipulation instructions perform logical operations on memory locations and registers.
7. Shift and rotate instructions displace the contents of a register or memory location.
8. The mode control instruction switches the processor between native and emulation mode.
9. Interrupt-related instructions include those that regulate requests for service from external devices.
10. Miscellaneous instructions are ones that don't fit in any other group.

Data Transfer Instructions

Data transfer instructions move information between registers, memory locations, and I/O devices. Table 4-3 summarizes these instructions in three groups: load and store, register transfer, and block move. Remember, *in emulation mode, the processor transfers bytes; in native mode, it can transfer bytes or words.* (Block moves always transfer blocks of bytes.)

Load and Store

The load and store instructions are the assembly language counterparts of PEEK and POKE in BASIC. A load operation reads or copies a memory value or immediate value into a register (either A, X, or Y).

Table 4-3

Mnemonic	Assembler Format	Flags							
		N	V	M	X	D	I	Z	C
<i>Load and Store</i>									
LDA	LDA source	*	*	.
LDX	LDX source	*	*	.
LDY	LDY source	*	*	.
STA	STA destination
STX	STX destination
STY	STY destination
STZ	STZ destination
<i>Register Transfer</i>									
TAX	TAX	*	*	.
TAY	TAY	*	*	.
TCD	TCD	*	*	.
TCS	TCS
TDC	TDC	*	*	.
TSC	TSC	*	*	.
TSX	TSX	*	*	.
TXA	TXA	*	*	.
TXS	TXS
TXY	TXY	*	*	.
TYA	TYA	*	*	.
TYX	TYX	*	*	.
XBA	XBA	*	*	.
<i>Block Move</i>									
MVN	MVN ss,dd
MVP	MVP ss,dd

Notes: (1) * means changed and . means unchanged.
 (2) Shaded instructions are new with the 65816.
 (3) *ss* and *dd* are the bank numbers for the source and destination respectively.

Two flags in the processor status register provide information about the value that has been loaded. The negative (N) flag is 1 if the value is negative and 0 if it is zero or positive. (Of course, N is only meaningful if you're working with signed numbers; you don't care about it for unsigned numbers.) The zero (Z) flag indicates whether the loaded value is zero ($Z = 1$) or something else ($Z = 0$).

For example, the following instruction loads the contents of memory location ThisLoc into the accumulator:

```
LDA ThisLoc
```

Again, N and Z reflect the sign of the loaded value and whether it is 0.

The store instructions copy the contents of the A, X, or Y register into a specified memory location. Unlike the load instructions, the stores do not affect the processor status register.

There is also STZ, which stores 0 in a location. The simplest way to store a value other than zero in memory is by using a load and store combination, such as:

```
LDA #$FFFE
STA ThatLoc
```

Since I'm discussing numbers in memory, it's worthwhile to mention that the 65816 stores 16-bit numbers in low-byte/high-byte order — that is, with the least-significant byte at the lower address. For example, when storing \$ABCD at a location called NUM, it puts \$CD at NUM and \$AB at the next location, NUM + 1. Keep this storage scheme in mind when you display the contents of memory. Just remember “low data, low address; high data, high address.”

Register Transfer

These instructions let you copy between two registers, in the combinations shown in Figure 4-1. Of them, only XBA (Exchange B and A Bytes of Accumulator) affects the contents of the source register.

Note that most of these transfers involve the accumulator (called A to be consistent with 6502 terminology or C to reflect its new 16-bit length in the 65816). That's because the accumulator is the only register on which the processor can perform arithmetic and logical operations. Hence, there are instructions that copy a value into the accumulator prior to an operation (e.g., TXA and TYA) and copy the result from the accumulator after it (TAX and TAY).

A note about the XBA instruction: The N and Z flags it returns reflect the state of the new A (low) byte.

Block Move

The block move instructions copy a block of bytes from one place in memory to another, working either forward (MVN) or backward (MVP) through the block.

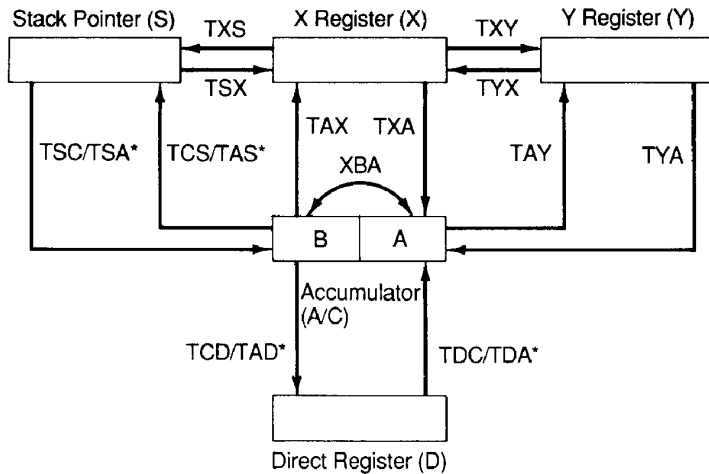


Figure 4-1

They have the general format:

```
MVN srcbk,destbk
MVP srcbk,destbk
```

where *srcbk* is the number of the bank that contains the original block (the source) and *destbk* is the number of the bank where the copy belongs (the destination). These numbers are the same if you're copying within a bank. Fortunately, you don't actually have to specify the bank numbers — simply enter the names of the blocks (e.g., OldBlock and NewBlock) and the assembler will calculate their bank numbers.

You must also supply the details of the operation in three registers. Specifically:

- The X register must contain the offset of the first byte to be copied.
- The Y register must contain the offset of the first copied byte.
- The accumulator must contain the number of bytes to be copied less 1.

Note that for MVN, X and Y must point to the beginning of the source and destination blocks, whereas for MVP, X and Y must point to the ends of those blocks.

If the source and destination blocks do not overlap, or if you're copying between banks, you can use either instruction. Make it easy on yourself. If the start addresses are easy to obtain, use MVN; if the end addresses are more convenient, use MVP. For example, to copy 100 bytes from location Here to location There, use:

```
LDX    #Here           ; Put source offset in X,
LDY    #There          ; destination offset in Y,
LDA    #99             ; and count - 1 in A
MVN    Here, There     ; Move the block
```

Note the use of the immediate mode to obtain the block offsets.

If the blocks overlap, it's critical to choose the correct block move instruction. For example, if you want to move a block forward one byte position in memory (to the next higher numbered address), and start at the beginning of the block, the first byte to be moved will overwrite the second byte. Instead, you must start moving from the end of the block, using MVP.

Similarly, if you move a block backward one byte position (to the next lower-numbered address), and start at the end of the block, the first byte moved will overwrite the preceding byte. In this case, you must start moving from the beginning of the block, using MVN. In summary:

- To move a block backward (to a lower address), put the start addresses of the source and destination in X and Y, then execute an MVN instruction.
- To move a block forward (to a higher address), put the end addresses of the source and destination in X and Y, then execute an MVP instruction.

Of course, in either case A must contain the byte count minus 1.

Figure 4-2 illustrates how the block move instructions operate when the source and destination blocks overlap. Here, the arrows inside the blocks indicate the sequence in which bytes are copied — from beginning to end for MVN, from end to beginning for MVP.

With MVN, the processor increments the X and Y registers each time it copies a byte; with MVP, it decrements X and Y each time; for both, it decrements the count in A. Hence, at the end of a block move operation, X and Y point to the byte that lies just beyond the last byte copied and A contains - 1 (hex FFFF). Moreover, the 65816 assumes that since you copied

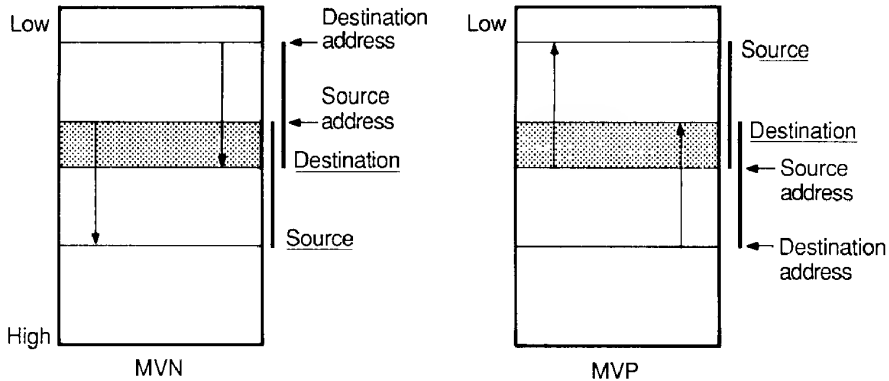


Figure 4-2

something to the destination bank, that's the bank you want to work with, so it makes the data bank register (DBR) point to that bank.

Because the block move instructions assume that X, Y, and A are 16-bit registers, you should only use them in the 65816's full native mode.

Arithmetic Instructions

The 65816 can operate in two different arithmetic modes, binary and decimal. When operating in binary mode (as it is when you switch the computer on), it treats operands as binary values; in decimal mode, it treats them as binary-coded decimal (BCD) numbers.

As Table 4-4 shows, there are instructions that add, subtract, increment, and decrement operands. There are also instructions that manipulate flags in the processor status register; these are used in various ways during arithmetic operations. Finally, there are compare instructions; these actually do a subtraction, but they report the results only in the status flags. Before discussing these instructions, I think it best to spend some time describing the formats of binary and decimal numbers.

Data Formats

As mentioned in chapter 0, binary numbers may be 8 or 16 bits long — depending on whether the 65816 is operating in emulation mode or native mode — and may be either unsigned or signed. In an *unsigned* number, all bits represent data. Therefore, unsigned numbers can range from 0 to 255 (8

Table 4-4

Mnemonic	Assembler Format	Flags							
		N	V	M	X	D	I	Z	C
<i>Flag Manipulation</i>									
CLC	CLC	0
CLD	CLD	0	.	.	.
CLV	CLV	.	0
SEC	SEC	1
SED	SED	1	.	.	.
<i>Add and Subtract</i>									
ADC	ADC source	*	*	*	*
SBC	SBC source	*	*	*	*
<i>Increment</i>									
INC	INC destination	*	*	.
INX	INX	*	*	.
INY	INY	*	*	.
<i>Decrement</i>									
DEC	DEC destination	*	*	.
DEX	DEX	*	*	.
DEY	DEY	*	*	.
<i>Compare</i>									
CMP	CMP source	*	*	*
CPX	CPX source	*	*	*
CPY	CPY source	*	*	*

Note: * means changed and . means unchanged.

bits) or 65,535 (16 bits). In a *signed* number, the high-order bit (7 or 15) specifies the sign of the number; the rest hold data. Therefore, signed numbers can range from 127 to -128 (bits) or from 32,767 to -32,768 (16 bits).

The 65816 can operate on decimal numbers that have been stored as a series of “packed” bytes. By packed, I mean that each byte holds two *binary-coded decimal (BCD)* digits, with the most-significant digit in the upper 4 bits. Therefore, a BCD byte can hold values from 00 to 99, while a BCD word can hold values from 0000 to 9999.

Addition

Most microprocessors have two add instructions: one that simply adds two operands and another that includes a carry in the addition. The intent here is that one would use the “add without carry” form to add single-byte or single-word operands or to add the low-order bytes or words of multiprecision

operands. On the other hand, one would use the “add with carry” form to add higher-order bytes of multiprecision operands.

Like the 6502, the 65816 has only one add instruction: *ADC (Add to Accumulator with Carry)*. Here, “Carry” is the carry (C) flag of the processor status register. ADC adds a source operand (an immediate value or the contents of a memory location) and the carry flag to the accumulator, and puts the result in the accumulator. In equation form, this is:

$$A = A + \text{source} + C$$

Since carry is always included, you must explicitly clear it to 0 before adding single-unit numbers or the low-order units of multiprecision numbers. The instruction that clears the carry flag is *CLC (Clear Carry Flag)*. For example, to add 15 to the accumulator, enter:

```
CLC           ; Clear the carry flag,
ADC #15       ; then add 15
```

ADC affects four flags in the processor status register:

- The negative (N) flag is 1 if the result is negative (the high-order bit is 1); otherwise, N is 0.
- The overflow (V) flag is 1 if adding two positive or negative numbers produces a result that exceeds the two’s-complement capacity of the accumulator, which changes the sign; otherwise, V is 0.
- The zero (Z) flag is 1 if the result is zero; otherwise, Z is 0.
- The carry (C) flag is 1 if the result cannot be contained in the accumulator; otherwise, C is 0.

N and V are only pertinent when you add signed numbers.

The 65816 has instructions that test flags and base an execution “decision” on the outcome. For example, a negative result (N = 1) may make it execute one set of instructions, while a zero or positive result (N = 0) makes it execute a different set. These decision-making instructions are discussed later in this chapter.

To add multiprecision numbers, clear the carry flag, add the low-order units, and store the result in memory. Then, to add the higher-order units, perform a series of LDA (load), ADC (add), and STA (store) instructions, one for each remaining unit. You can use the same procedure to add multiprecision decimal (BCD) numbers; simply precede the addition with an *SED (Set Decimal Mode)* instruction.

How the 65816 Subtracts

Like other general-purpose microprocessors, the 65816 has an internal addition unit, but no subtraction unit. Still, it *can* subtract numbers — by adding them! Strange as this may seem, the concept is “elementary,” as Sherlock Holmes might say.

To see how to subtract numbers by adding them, consider how you subtract, say, 7 from 10. In elementary school, you learned to write this as:

$$10 - 7$$

However, later (in Algebra 101, perhaps) you learned that another way to write it is:

$$10 + (-7)$$

The first form — the straight subtraction — can be performed by a processor that has a subtraction unit. Since the 816 has no such unit, it subtracts in two steps. First, it changes the sign of, or *complements*, the second number (the subtrahend). Then it adds the complemented subtrahend to the minuend (the first number) to produce the result. Because the 816 works with base 2 (binary) numbers, the complement is a *two's-complement*. To obtain the two's-complement of a binary number, take its positive form and reverse each bit — change each 1 to 0 and each 0 to 1 — then add 1 to the result.

Applying this to the “10 - 7” example, the 8-bit binary representation of 10 and 7 are 00001010 and 00000111, respectively. Take the two's-complement of 7 as follows:

$$\begin{array}{rcl} 1111\ 1000 & \text{(Reverse all bits)} & \\ + \quad \quad 1 & \text{(Add 1)} & \\ \hline 1111\ 1001 & \text{(Two's complement of 7, or -7)} & \end{array}$$

Now the subtraction operation becomes:

$$\begin{array}{rcl} 0000\ 1010 & (= 10) & \\ + 1111\ 1001 & (= -7) & \\ \hline 0000\ 0011 & (= 3) & \end{array}$$

Eureka! That's the right answer!

Subtraction

As in the case of addition, the 65816 has only one subtract instruction, *SBC* (*Subtract from Accumulator with Borrow*), in which the carry (C) flag contributes the “borrow.” SBC subtracts a source operand (an immediate value or the contents of a memory location) and the inverse, or *complement*, of the carry flag from the accumulator, and puts the result in the accumulator. In equation form, this is:

$$A = A - \text{source} + C$$

Since carry is always included, you must explicitly set it to 1 before subtracting single-unit numbers or the low-order units of multiprecision numbers. The instruction that sets the carry flag is *SEC* (*Set Carry Flag*). For example, to subtract 15 from the accumulator, enter:

```
SEC           ;Set the carry flag
SBC  #15      ; then subtract 15
```

SBC affects four flags in the processor status register:

- The negative (N) flag is 1 if the result is negative (the high-order bit is 1); otherwise, N is 0.
- The overflow (V) flag is 1 if you subtract a positive number from a negative, or vice versa, and the result exceeds the two’s-complement capacity of the accumulator, which changes the sign; otherwise, V is 0.
- The zero (Z) flag is 1 if the result is zero; otherwise, Z is 0.
- The carry (C) flag is 1 if the result is positive or zero; C is 0 if the result is negative, indicating a borrow.

N and V are only pertinent when you add signed numbers.

To subtract multiprecision numbers, set the carry flag, subtract the low-order units, and store the result in memory. Then, to subtract the higher-order units, perform a series of LDA (load), SBC (subtract), and STA (store) instructions, one for each remaining unit. You can use the same procedure to subtract multiprecision decimal (BCD) numbers; simply precede it with an *SED* (*Set Decimal Mode*) instruction.

Signed Arithmetic

I haven't yet mentioned how adding and subtracting differs between signed and unsigned numbers. That was not an oversight; I didn't mention it because you use the same instructions, ADC and SBC, regardless of whether the operands are signed or unsigned. However, for signed numbers, you must pay attention to the states of the negative (N) and overflow (V) flags.

The N flag simply reflects the sign of the result (positive if N = 0, negative if N = 1), but the V flag indicates whether the result in the accumulator is valid (V = 0) or invalid (V = 1). Recall that V is 1 if adding two numbers having the same sign or subtracting two numbers having different signs produces a result that exceeds the capacity of the accumulator.

Generally, your program should check for overflow after each signed add or subtract operation, and run some kind of error routine (display a message, perhaps) if V is 1. I'll describe instructions that test for overflow later in this chapter. Once set, the overflow flag stays that way until you either begin another ADC or SBC operation or execute a *CLV* (*Clear Overflow Flag*) instruction, which resets V to 0.

Increment and Decrement

The 65816 has a group of instructions that increment or decrement an operand — that is, add 1 to it or subtract 1 from it. The *INC* and *DEC* instructions require an operand, either A (for accumulator) or one of the memory addressing modes.

INX, *INY*, *DEX*, and *DEY* have no operand; the X or Y register is implied in the instruction. These are convenient for increasing or decreasing loop counters when you're doing a repetitive operation (e.g., adding a table of numbers in memory). The X and Y forms are particularly useful for increasing or decreasing the index register when you are accessing consecutive locations in memory.

Example 4-1 illustrates both of these uses; it is a short program that fills a 100-element table (ZTable) with zeros. Here, X serves as the element pointer and Y holds the count. The final instruction, BNE, is one I haven't described yet, but all it does is make the processor continue looping back to NextEel until Y has been decremented to 0. Note also that I have used *STZ* (*Store Zero*), an instruction introduced with the 65C02, to store the zeros. With a 6502, this would require two instructions: an *LDA #0* preceding NextEel and a *STA ZTable,X* at NextEel.

The increment and decrement instructions are specialized adds and

Example 4-1

```

; Fill a 100-element table called ZTable with zeros.

      LDX  #0          ;Index points to first element
      LDY  #100        ;Operate on 100 elements
NextEl STZ  ZTable,X    ;Store zero in next element
      INX          ; and increase the pointer
      DEY          ;Decrease the count
      BNE  NextEl      ;Loop until count is zero

```

subtracts that affect only the negative (N) and zero (Z) flags; they leave the overflow (V) and carry (C) flags alone. This is important because it lets you, for example, use a loop to add multiprecision numbers, yet preserve the carry between operations.

Compare

Most programs don't execute instructions in the order they are stored in memory. Instead, they usually include jumps, loops, and subroutine calls that make the 816 transfer to a different part of a program. The instructions that actually produce these transfers will be discussed in the next section, "Control Transfer Instructions." But I will now discuss the compare instructions, which help the control transfer instructions make their transfer/no-transfer "decisions."

The *CMP*, *CPX*, and *CPY* instructions do a "trial" subtraction. That is, they subtract a source operand (an immediate value or the contents of a memory location) from a destination operand (A, X, or Y), but *do not save the result*. That is, they don't alter the destination. Instead, they only set or clear the flags based on the result (see Table 4-5). Further, unlike *SBC*, the compares do not include the carry flag in the subtraction.

Table 4-5

Condition	N*	Z	C
A, X, or Y < Memory	1	0	0
A, X, or Y = Memory	0	1	1
A, X, or Y > Memory	0	0	1

*Meaningful only for operations on signed numbers.

Control Transfer Instructions

The control transfer instructions can make the 65816 transfer from one part of a program to another. All but the simplest programs include jumps and subroutine calls that alter the execution path the microprocessor takes. Subroutine instructions are discussed in the next section; this section covers jumps and branches. Table 4-6 divides the control transfer instructions into two groups: unconditional transfer and conditional transfer. Note that none of these instructions affect the flags.

Unconditional Transfer

There are two kinds of unconditional transfer instructions, jumps and branches. The *JMP* instruction is the assembly language equivalent of GOTO in BASIC; it makes the 65816 take its next instruction from some place other than the next consecutive memory location. *JMP* has the general form

JMP target

Table 4-6

Mnemonic	Assembler Format	Flags							
		N	V	M	X	D	I	Z	C
<i>Unconditional Transfer</i>									
JMP	JMP target
JML	JML (aaaa)
BRA	BRA target
BRL	BRL long target
<i>Conditional Transfer</i>									
BCC	BCC target
BCS	BCS target
BEQ	BEQ target
BMI	BMI target
BNE	BNE target
BPL	BPL target
BVC	BVC target
BVS	BVS target

Notes: (1) . means unchanged.
 (2) Shaded instructions are new with the 65816.

where *target* is a location (usually a label) in the active program bank. JMP is often used to bypass a group of instructions that are executed from some other part of the program. For example, you may use it in this context:

```

        . .
        . .
        LDA      DATA1
        CLC
        ADC      DATA2
        JMP      THERE
HERE    . .
        . .
THERE   . .
        . .

```

You can also jump to a location *indirectly*, by obtaining its two-byte address from a pointer in memory. For example,

```
JMP      (MEMPTR)
```

jumps to the location whose address is in MEMPTR.

To transfer to an instruction in a different bank, you must use *JML* (*Jump Long*). JML uses only absolute indirect address, but unlike JMP, obtains a 3-byte address — the program bank register (PBR) and the program counter (PC) — from a memory pointer. JMP is either a 3- or 4-byte instruction, depending on whether the target is in the same bank or a different one. JML is always a 3-byte instruction.

If the target address is no more than 127 bytes away, you can use a faster version of JMP: *BRA* (*Branch Always*). Similarly, if the target is no further than 32,767 bytes, you can use *BRL* (*Branch Always Long*).

Conditional Transfer

There are eight *branch* instructions that let the 65816 make an execution “decision” based on some prescribed condition, such as a register containing 0 or the carry (C) flag being set to 1. If the condition is satisfied, the 816 makes the transfer; otherwise, it continues to the next instruction. Table 4-7 summarizes the branch instructions and the flags they test.

As you can see from Table 4-7, the branch instructions base their transfer decisions on the contents of four flags in the processor status register: carry (C), zero (Z), negative (N), and overflow (V). In general:

Table 4-7

Instruction	Description	Branch if . . .
BCC/BLT	Branch on Carry Clear/if Less Than	Carry (C) = 0
BCS/BGE	Branch on Carry Set/if Greater Than or Equal to	Carry (C) = 1
BEQ	Branch if Equal	Zero (Z) = 1
BNE	Branch if Not Equal	Zero (Z) = 0
BMI	Branch if Minus	Negative (N) = 1
BPL	Branch if Plus	Negative (N) = 0
BVC	Branch on Overflow Clear	Overflow (V) = 0
BVS	Branch on Overflow Set	Overflow (V) = 1

- The carry flag reflects conditions where the result cannot be contained in the result register or memory location. It is affected indirectly by the arithmetic, shift, and compare instructions, and directly by SEC and CLC.
- The zero and negative flags are set by conditions in which the result is 0 or has the most-significant bit equal to 1. These flags can be affected by about half of the 65816 instructions.
- The overflow flag is affected by the arithmetic instructions (ADC and SBC) and by the BIT and CLV instructions.

The branch instructions use only program counter relative addressing. The relative displacement is a signed byte contained in the instruction, so the branch can span up to 127 bytes forward or 128 bytes backward from the instruction that follows the branch. This is not as restricting as you might think, because you can always combine a branch instruction with a JMP or JML to transfer anywhere in memory. For example, here is how a program might branch on the carry-set condition to an instruction (at CSET) that is beyond the normal +127/−128 branch range:

```

                BCC    CCLEAR    ;Go to CCLEAR on carry = 0
                JMP    CSET      ;Go to CSET on carry = 1
CCLEAR         . .
                . .

```

The conditional branch instructions each occupy 2 bytes in memory: opcode followed by the relative displacement. The 65816 takes 2 cycles to execute a branch if the condition is not met and 3 cycles if the condition is

met (4 cycles if the 816 crosses a page boundary). Because of the time difference, construct your programs so that whenever possible, the expected case executes if the branch is *not taken*.

Here are some examples of branch instructions:

1. The following sequence branches to TOOBIG if the addition produces a carry.

```
ADC    MEMLOC
BCS    TOOBIG
```

2. This sequence branches to TOOSML if the subtraction produces a borrow.

```
SBC    MEMLOC
BCC    TOOSML
```

3. These instruction will branch to ZERO if A and MEMLOC hold the same value.

```
CMP    MEMLOC
BEQ    ZERO
```

4. The following sequence will loop to LOOP until the X register has been decremented to 0. This sort of sequence is common in programs that use the X or Y register as a counter.

```
LOOP    . .
        . .
        DEX
        BNE    LOOP
```

Using Branch Instructions with Compares

You can precede conditional branch instructions with any instruction that alters the flags, but they are often preceded with a compare instruction (CMP, CPX, or CPY). Table 4-5 earlier in this chapter shows how the compares affect the flags for various register/memory combinations.

Now, with the variety of conditional branch instructions, it is worthwhile to look at a more practical table — one that shows which conditional branch to use for all possible register/data combinations. Table 4-8 is the one

Table 4-8

To branch if . . .	Follow compare with			
	For unsigned numbers		For signed numbers	
Register is less than data	BLT	THERE	BMI	THERE
Register is equal to data	BEQ	THERE	BEQ	THERE
Register is greater than data	BEQ	HERE	BEQ	HERE
	BCS	THERE	BPL	THERE
Register is less than or equal to data	BLT	THERE	BMI	THERE
	BEQ	THERE	BEQ	THERE
Register is greater than or equal to data	BGE	THERE	BPL	THERE

you need. In this table, THERE represents the label of the instruction the 65816 executes if the branch test succeeds, while HERE is the label of the instruction the 816 executes if the test fails.

To illustrate a typical application for a compare/branch combination, Example 4-2 shows a program that arranges two unsigned numbers in memory in increasing order, with the larger number in the higher-numbered location.

You can also combine a compare instruction with two branch instructions to test the “less than,” “equal to,” and “greater than” cases separately. Example 4-3 shows a sequence that executes any of three groups of instructions, based on whether the value in A is below, equal to, or above 10. Since the branch instructions do not affect the status flags, BNE GT10 can base its branch decision on the same flag that the BGE GTEQ10 based *its* decision on.

Example 4-2

```
; This sequence arranges two unsigned 16-bit numbers in memory
; in order of magnitude, with the larger value at the higher
; address.
```

```
        LDA  MEMLOC+2  ;Get second value
        CMP  MEMLOC     ;Compare the numbers
        BGE  DONE       ;Done if second is greater than
                        ; or equal to the first
        LDX  MEMLOC     ;Otherwise, swap them
        STA  MEMLOC
        STX  MEMLOC+2
DONE    ..
        ..
```


Example 4-3

; This sequence executes one of three different groups of
 ; instructions, based on whether the unsigned number in A
 ; is below, equal to, or above 10.

```

                CMP  #10      ;Compare accumulator to 10
                BGE  GTEQ10   ;Accumulator is less than 10
                ..
                BRA  DONE
GTEQ10         BNE  GT10      ;Accumulator is equal to 10
                ..
                BRA  DONE
GT10          ..            ;Accumulator is greater than 10
                ..
DONE          ..
                ..

```

Subroutine Instructions

Sometimes you want to perform a specific operation (say, display a message) at more than one place in your program. One way to do that is to duplicate the entire set of instructions everywhere you need it. However, duplicating instructions is both frustrating and time-consuming. It also makes programs longer than they would be if you could avoid this duplication. As a matter of fact, you *can* eliminate needless duplication by defining a recurring instruction sequence as a *subroutine*. Table 4-9 lists the 65816's subroutine instructions.

As in BASIC, a subroutine is a set of instructions that you write just once, but which you can execute as needed at any place in a program. The

Table 4-9

Mnemonic	Assembler Format	Flags							
		N	V	M	X	D	I	Z	C
JSR	JSR target
RTS	RTS
JSL	JSL long-target
RTL	RTL

Notes: (1) means unchanged.
 (2) Shaded instructions are new with the 65816.

process of transferring control from the main program to a subroutine is defined as *calling*. When you *call* a subroutine, the 65816 executes the instructions in it, then returns to the place from which the call was made.

This invites two questions: “How does one call a subroutine?” and “How does the 816 return to the proper place in the program?” The answers involve two subroutine-related instructions: JSR and RTS.

Instructions that execute subroutines must perform three tasks:

1. They must somehow save the contents of the program counter (PC). Once the subroutine has been executed, the 816 uses this address to return to the calling point. Hence, we refer to the saved address as the *return address*.
2. They must make the microprocessor begin executing the subroutine.
3. Upon completion of the subroutine, they must use the stored value of the PC to return to the main program and continue executing at that point.

These three tasks are performed by two instructions: *JSR* (*Jump to Subroutine*) and *RTS* (*Return from Subroutine*). Essentially, JSR and RTS are the assembly language equivalents of GOSUB and RETURN in BASIC.

JSR performs tasks 1 and 2; it stores the return address and begins executing. Specifically, it stores the return address (the address of the instruction that follows it) on the stack. JSR has the format

```
JSR  target
```

where *target* is the name of the subroutine being called.

RTS undoes the work of JSR; that is, it makes the 816 leave the subroutine and return to the calling program by pulling the return address off the stack. RTS must always be the last subroutine instruction the processor executes. (This doesn’t mean that RTS must be the last instruction in the subroutine — although it usually is — just the last one the 816 *executes*.)

For example, to call a subroutine named MYSUB, your program might execute this sequence (offsets are also listed):

```
04F0          JSR MYSUB    ;Call the subroutine
04F3 NEXT    TXA          ;Return here after the subroutine
```

```

0600 MYSUB LDA #6      ;First instruction of the
                        ;subroutine
      .
      .
061E      RTS          ;Return to calling program

```

When the 65816 executes JSR MYSUB, it pushes the offset of NEXT onto the stack, then loads the offset of MYSUB into the program counter (PC) — and that's where the 816 begins executing. Eventually, when the 816 encounters the RTS instruction, it pulls the return address off the stack and puts it into the PC. This makes it resume at the instruction labeled NEXT, a TXA in this case. Figure 4-3 shows the stack, the stack pointer (S), and the program counter (PC) before and after the JSR, and after the RTS.

The 65816 also provides two instructions, *JSL* (*Jump to Subroutine Long*) and *RTL* (*Return from Subroutine Long*), for subroutines that are in a different bank than the JSR instruction. JSL pushes 3 bytes (program bank register and program counter) onto the stack, as opposed to 2 bytes (PC) for JSR.

A subroutine may itself call other subroutines. For example, a subroutine that reads a user's menu response from the keyboard may well decode the response character and then call one of several other subroutines based on the result. Calling one subroutine from within another is referred to as *nesting*. Figure 4-4 shows the JSR and RTS instructions for a program in which SUBR1 calls SUBR2 (i.e., SUBR2 is nested within SUBR1).

Programmers usually describe nesting in terms of *levels*. An application like the one in Figure 4-4, where the nesting extends only to the JSR to SUBR2 (SUBR2 does not call another subroutine) is said to have one level of nesting. However, SUBR2 might well have called a third subroutine — say, SUBR3 — with SUBR3 calling SUBR4, and so on.

Stack Instructions

As mentioned in the preceding section, the stack holds return addresses while the 65816 is executing subroutines. The JSR (or JSL) instruction puts the address onto the stack and an RTS (or RTL) instruction retrieves it at the conclusion of the subroutine. In both cases, the processor uses the stack *automatically*; you don't have to tell it to do so.

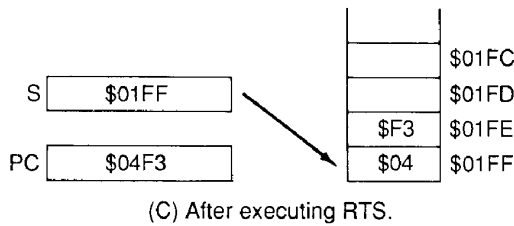
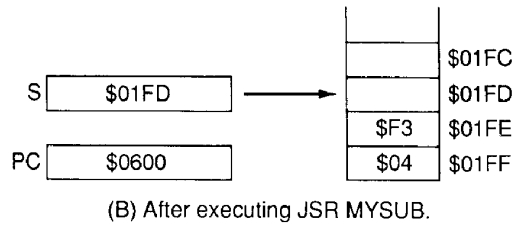
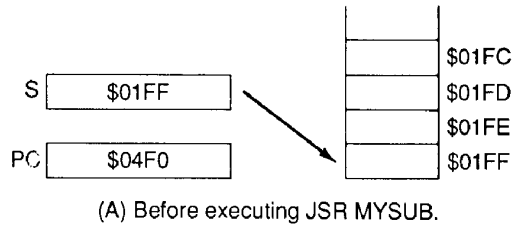


Figure 4-3

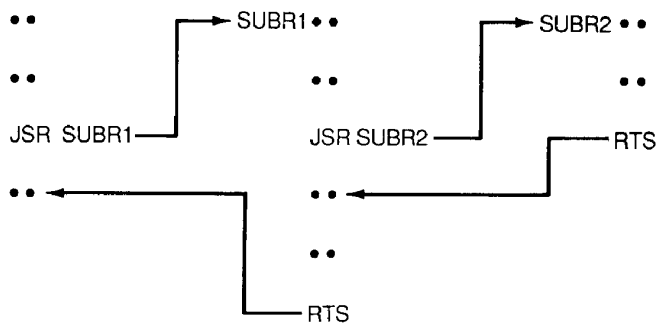


Figure 4-4

The stack is also a convenient place to deposit data from your program temporarily. For example, you might want to save the contents of the accumulator while you put it to some other use.

Overview of the Stack

Earlier, I mentioned that the stack is a “last-in/first-out” type of data structure in page 1 of memory. That is, the last item to be entered (or *pushed*) onto the stack is the first item to be extracted (or *pulled*) from it. Conversely, the first item pushed onto the stack is the last item to be pulled off it. In short, *stack data is retrieved in the opposite order from which it was stored*, just like plates in a kitchen cabinet.

Stack information is accessed by a dedicated stack address register called the *stack pointer (S)*, which always points to the next available location on the stack. The 65816 decrements the stack pointer whenever a byte is pushed onto the stack, and increments it whenever a byte is pulled from the stack. Hence, the stack “builds” downward in memory, in the direction of location 0. When you switch the computer on, the stack pointer points to location \$01FF, the end of page 1.

Among the data transfer instructions discussed earlier in this chapter, I described several that are used to copy between the stack pointer and another register: TCS, TSB, TSC, TSX, and TXS. You wouldn’t normally use these instructions, however, because for most applications, you simply let the 65816 take care of regulating the stack pointer.

Table 4-10 shows the more important stack instructions, the ones you use to push information onto the stack and pull information off it. These instructions are divided into two groups: *push and pull registers* and *push immediate data or effective address*.

Push and Pull Registers

As Table 4-10 shows, the 65816 provides instructions that push and pull the contents of the accumulator (*PHA* and *PLA*), data bank register (*PHB* and *PLB*), direct register (*PHD* and *PLD*), processor status register (*PHP* and *PLP*), and the X (*PHX* and *PLX*) and Y (*PHY* and *PLY*) registers. It also has a *PHK* instruction that pushes the contents of the program bank register.

Aside from the fact that they operate on different registers, the push instructions work identically. In each case, the 816 pushes the contents of the register onto the stack at the location to which the stack pointer is pointing. Then it decrements the stack pointer by 1 (in emulation mode) or 2 (in native mode), making it point to the next lower location. Push instructions

Table 4-10

Mnemonic	Assembler Format	Flags							
		N	V	M	X	D	I	Z	C
<i>Push and Pull Registers</i>									
PHA	PHA
PHB	PHB
PHD	PHD
PHK	PHK
PHP	PHP
PHX	PHX
PHY	PHY
PLA	PLA	*	*	.
PLB	PLB	*	*	.
PLD	PLD	*	*	.
PLP	PLP	*	*	*	*	*	*	*	*
PLX	PLX	*	*	.
PLY	PLY	*	*	.
<i>Push Immediate Data or Effective Address</i>									
PEA	PEA Loc
PEI	PEI (DLoc)
PER	PER #Num

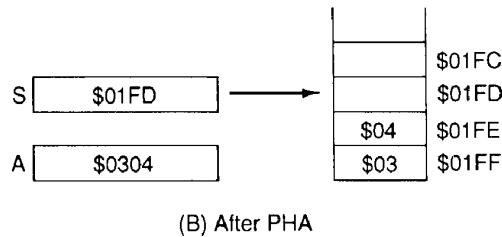
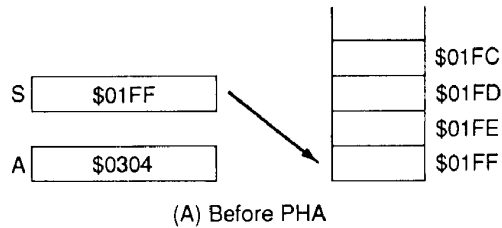
Notes: (1) * means changed and . means unchanged.
 (2) Shaded instructions are new with the 65816.

do not alter the contents of the source register, nor do they affect the status flags.

Figure 4-5 shows how a PHA instruction in 16-bit native mode affects the stack. Here, the stack pointer (S) is pointing to the top of the stack, location \$01FF, and the accumulator contains \$0304. After PHA executes, the stack pointer has been decremented to \$01FD and the contents of the accumulator have been stored on the stack.

If the PHA was followed by a PLA instruction, the 65816 would do the reverse; it would increment S, copy the contents of location \$01FE into the low byte of the accumulator, increment S again, and copy the contents of location \$01FF into the high byte of the accumulator. (Note that while \$0304 is still stored in the same location in memory, it is no longer considered as being “on the stack.” In fact, *nothing* is on the stack, because the stack pointer is pointing at the top of the stack, location \$01FF.)

With seven push and six pull instructions at your disposal, you can save

**Figure 4-5**

all the registers if you want to. That is, you can save the entire “context” in which your program is working. Example 4-4 shows the instructions you would use to save and restore all the general registers and the status. These are the kinds of instructions you should put at the beginning and end of any subroutine that is to save the caller’s context.

Note that I have entered the pull instructions in the opposite order from the push instructions. This reflects the “last-in/first-out” nature of the stack. Having mentioned that, I must state the cardinal rule for working with the stack:

Example 4-4

; Use these instructions to preserve the accumulator, X, Y,
; and status.

```

PHP          ;Save processor status register,
PHA          ; accumulator,
PHX          ; X register,
PHY          ; and Y register
..
..
PLY          ;Restore the Y register,
PLX          ; X register,
PLA          ; accumulator,
PLP          ; and status

```

You must always pull data in the reverse order you pushed it.

Note also that I pushed the processor status register first. This lets me pull it last, to cancel the effects of the other pull instructions on the N and Z flags (see Table 4-10).

While I'm giving rules, there's another one you should remember:

Every push must have a corresponding pull later in the program.

By keeping these two rules in mind, you shouldn't have any problems with stack operations.

Later in this book, you will encounter programs that start with the instruction combination:

```
PHK ; Copy the contents of the program bank register
PLB ; into the data bank register
```

This is necessary because the system loader makes the PBR point to the bank that contains your program, but it does nothing to the DBR. The PHK/PLB combination tells the microprocessor that data within the program resides in the same bank as the program itself. You might assume that the loader would do this for you, but it does not; at least not in the current version of the software.

Push Immediate Data or Effective Address

The designers of the 65816 included an instruction called *PEA*, which is short for *Push Effective Address onto Stack*. However, it should really be described as "Push Immediate Word onto Stack" (and named, perhaps, PIW), because that's what it does; it pushes a 16-bit constant onto the stack. For example,

```
PEA #15
```

pushes decimal 15 onto the stack. Of course, there's no "Pull Immediate Data" instruction, so you have to pull it into a register.

Another misnamed instruction is *PEI*, short for *Push Effective Indirect Address onto Stack*. PEI should be described as "Push Direct Page Word onto Stack" (and named, say, PDW), because that's what it does. It takes a

byte operand from the instruction and uses it as an offset into the direct page. PEI forms the effective address by adding the offset to the direct register (D), then pushes the word at that location onto the stack. For example,

```
PEI    #4
```

pushes the word that starts at byte 4 of the direct page onto the stack.

The final instruction of this group, *PER (Push Effective Program Counter Relative Address onto Stack)*, takes a 16-bit offset from the instruction and adds it to the value of the program counter, then pushes the result onto the stack. PER does *not* change the program counter or the program bank register.

Bit Manipulation Instructions

These instructions manipulate bit patterns in the accumulator or a memory location. Table 4-11 divides them into three groups: logical, bit testing, and processor status bits.

Logical

Logical instructions are so named because they operate according to the rules of formal logic, rather than those of mathematics. For example, the rule of logic that states, “If A is true and B is true, then C is true” has a 65816 counterpart in the *AND (AND Memory with Accumulator)* instruction. AND applies this rule to corresponding bits in two operands; one operand is the accumulator, the other can be an immediate value or the contents of a memory location.

Specifically, for each bit position where both operands are 1 (true), AND sets the bit in the accumulator to 1. Conversely, for any bit position where the two operands have any other combination — both are 0 or one is 0 and the other is 1 — AND sets the accumulator bit to 0 (see Table 4-12).

In essence, AND masks out (zeros) certain bits so you can do some kind of processing on the remaining bits. Note that *any bit ANDed with 0 becomes 0, and any bit ANDed with 1 retains its original value*. For example, this instruction zeros the high byte of the accumulator:

```
AND    #$00FF    (or simply AND    #$FF)
```

Table 4-11

		Flags						
Mnemonic	Assembler Format	V	M	X	D	I	Z	C
<i>Logical</i>								
AND	AND source	*	*	.
EOR	EOR source	*	*	.
ORA	ORA source	*	*	.
<i>Bit Testing</i>								
BIT	BIT source-byte	M ₇	M ₆	.	.	.	*	.
BIT	BIT source-word	M ₁₅	M ₁₄	.	.	.	*	.
BIT	BIT #Num	*	.
TRB	TRB destination	*	.
TSB	TSB destination	*	.
<i>Processor Status Bits</i>								
REP	REP #dd	*	*	*	*	*	*	*
SEP	SEP #dd	*	*	*	*	*	*	*

Notes: (1) * means changed and . means unchanged.
 (2) Shaded instructions are new with the 65816.

Table 4-12

Source	Destination	Result		
		AND	ORA	EOR
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

As in this example, you generally use hexadecimal numbering for an immediate operand. Since the 816 can operate on either bytes or words, you normally deal with either 2 or 4 hexadecimal digits. To help you construct the correct "mask" value for a logical operation, Table 4-13 shows the hexadecimal representation of a 1 in 16 different bit positions. For example, to operate on bit 2, the correct mask value is 4; to operate on bits 2 and 3, the mask is \$C (\$4 + \$8); and so on.

AND is also useful for converting digits that an operator types at the keyboard into binary numbers that your program can process. Typed digits

Table 4-13

Bit Number	Hex. Value	Bit Number	Hex. Value
0	0001	8	0100
1	0002	9	0200
2	0004	10	0400
3	0008	11	0800
4	0010	12	1000
5	0020	13	2000
6	0040	14	4000
7	0080	15	8000

— or characters, for that matter — enter the computer in a form called *ASCII* (short for American Standard Code for Information Interchange). As the ASCII summary in Appendix B shows, the digits 0 through 9 have the ASCII codes \$30 through \$39, respectively. To convert the code to its binary value, simply chop off the 3. With the code in the low byte of the accumulator, this requires:

AND #\$000F (or simply AND #\$F)

The *ORA (OR Memory with Accumulator)* instruction produces a 1 in the accumulator for each bit position in which either or both operands contain 1 (see Table 4-12). ORA is generally used to set specific bits to 1. For example,

ORA #\$C000

sets the accumulator's two high-order bits (14 and 15) to 1 and leaves all other bits unchanged.

The *EOR (Exclusive-OR Memory with Accumulator)* can be used to determine which bits differ between two operands or to reverse the setting of selected bits. EOR puts a 1 in the accumulator for every bit position in which the operands differ — that is, where one operand has 0 and the other has 1. If both operands' bits are the same (0 or 1), EOR clears the accumulator bit to 0. For example,

EOR #\$C000

reverses the state of the accumulator's two high-order bits (14 and 15) and leaves all other bits as they are.

As Table 4-12 shows clearly, EOR is the same as ORA, except that two 1's produce a 0 rather than a 1. EOR excludes the two 1's from the combinations that produce a 1 result — that's why it's called *Exclusive-OR*.

Bit Testing

The *BIT (Bit Test)* instruction ANDs a memory or immediate operand with the value in the accumulator, but affects only the flags, not the accumulator. With an immediate operand, BIT reports the result of the AND operation only in the zero (Z) flag. With a memory operand, BIT affects three status flags, as follows:

- The negative (N) flag receives the initial (un-ANDed) value of bit 7 (byte) or bit 15 (word) of the memory location.
- The overflow (V) flag receives the initial value of bit 6 (byte) or bit 14 (word) of the memory location.
- The zero (Z) flag is 1 if the AND operation produces a zero result; otherwise, Z is 0.

Note that only the Z flag reflects the result of the AND operation; N and V simply report the state of the operand's two high-order bits. If you follow BIT with a BNE (Branch if Not Equal) instruction, the 816 makes the branch if there are any corresponding 1 bits in both operands.

TRB (Test and Reset Bits) and *TSB (Test and Set Bits)* let you set or reset selected bits in a memory location. (In reality, "Test" is a misnomer. These instructions don't test the operand; they set or reset bits unconditionally.) TRB and TSB are useful for manipulating locations that act as indicators, where individual bits are meaningful.

The TRB instruction ANDs the memory operand with the complement of the accumulator (that is, with the accumulator's inverse). Thus, each 1 bit in A resets the corresponding memory bit to 0 and each 0 bit leaves its memory counterpart as it is. For example, if the accumulator contains 9,

```
TRB  MEMFLAG
```

clears bits 0 and 3 of MEMFLAG.

The TSB instruction ORs the memory operand with the accumulator. Thus, each 1 bit in the accumulator sets the corresponding memory bit to 1 and each 0 bit leaves its memory counterpart as it is. For example, if the accumulator contains \$40,

TSB MEMFLAG

sets bit 6 of MEMFLAG.

Processor Status Bits

Under “Arithmetic Instructions,” I described instructions that clear the overflow flag (CLV) and set or clear the carry and decimal mode bits (CLC, CLD, SEC, and SED). There are also instructions that manipulate the IRQ disable bit, and I’ll discuss them later. Now I will describe two instructions that let you manipulate the entire processor status register, rather than just individual bits. They are similar to TRB and TSB, except they operate on the status register rather than on a memory location.

The *REP (Reset Processor Status Bits)* instruction ANDs the status register with the complement (i.e., the inverse) of an immediate byte. Thus, each 1 bit in the byte resets the corresponding status bit to 0 and each 0 bit leaves its status counterpart as it is. For example,

REP #%110000

clears the M and X bits, thereby making the accumulator and index registers 16 bits long. (You should follow this particular REP with the directives LONGA ON and LONGI ON, to make the assembler assemble for the long registers.)

The *SEP (Set Processor Status Bits)* instruction ORs the status register with an immediate byte. Thus, each 1 bit in the byte sets the corresponding status bit to 1 and each 0 bit leaves its status counterpart as it is. For example,

SEP #%110000

sets the M and X bits, thereby making the accumulator and index registers 8 bits long. (Follow this SEP with LONGA OFF and LONGI OFF, to assemble for the short registers.) You will see these M- and X-changing instructions later, when the mode control instruction, XCE, is discussed.

You can also use SEP to set the overflow (V) flag — say, to use it as a 1-bit indicator in a program. The 65816 has no other instruction that sets V directly. The form you would use is:

SEP #%1000000

Shift and Rotate Instructions

The 65816 has four instructions that displace the contents of the accumulator or a memory location one bit position to the left or right (see Table 4-14). Two of these instructions *shift* the operand, the other two *rotate* it.

For all four instructions, the carry (C) flag acts as a “9th bit” or “17th bit” extension of the operand. That is, C receives the value of the bit that has been displaced out of the operand. A right shift or rotate puts the value of bit 0 into C; a left shift or rotate puts the value of bit 7 (byte) or bit 15 (word) into it. Figure 4-6 shows how these instructions operate.

Shifts

ASL (Arithmetic Shift Left) shifts the operand one bit position to the left and puts a 0 in the vacated bit 0 position. Similarly, *LSR (Logical Shift Right)* shifts the operand one bit position to the right and puts a 0 in the vacated high-order bit position (bit 7 in a byte, bit 15 in a word). Besides carry, ASL and LSR update the negative and zero flags.

To see how the shift instructions work, assume the processor is in 8-bit mode and the accumulator contains \$B4, or binary 10110100. Here is how the shift instructions affect A and C:

After ASL A: (A) = 01101000 C = 1

After LSR A: (A) = 01011010 C = 0

The shift instructions can also serve as multiply-by-2 or divide-by-2

Table 4.14

Mnemonic	Assembler Format	Flags							
		N	V	M	X	D	I	Z	C
<i>Shifts</i>									
ASL	ASL destination	*	*	*
LSR	LSR destination	0	*	*
<i>Rotates</i>									
ROL	ROL destination	*	*	*
ROR	ROR destination	*	*	*

Note: * means changed and . means unchanged.

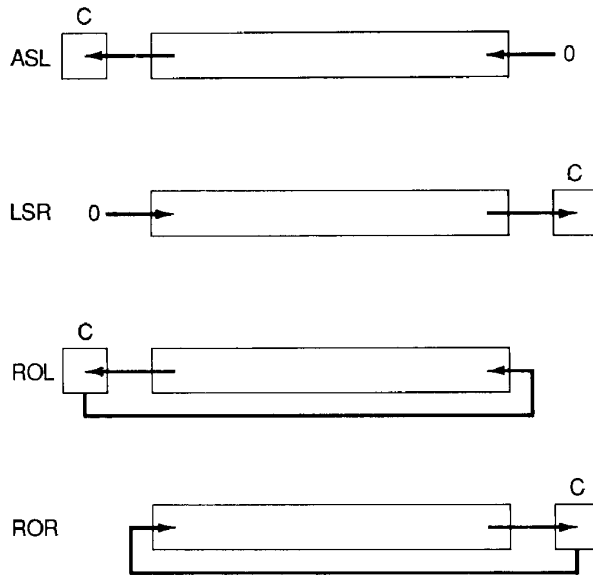


Figure 4-6

instructions, because *shifting an operand one bit position to the left doubles its value and shifting it one bit position to the right halves its value*. Of course, this assumes that the bit you shift out is a 0 rather than a 1.

Rotates

Like the shifts, the rotate instructions, *ROL (Rotate Left)* and *ROR (Rotate Right)*, enter displaced bits into the carry (C) flag. However, the rotates first enter the prerotate value of C into the vacated bit position at the opposite end of the operand. Like the shifts, the rotates also report the result in the negative and zero flags.

To see how the rotate instructions work, consider the operand that was used for the shift examples, \$B4 or binary 10110100, and assume that C is 1 initially. Here is how the two rotate instructions affect the accumulator and C:

After ROL A: (A) = 01101001 C = 1

After ROR A: (A) = 11011010 C = 0

Shifting Signed Numbers

Despite the fact that ASL stands for “Arithmetic Shift Left,” the four shift and rotate instructions perform what are known as *logical* shifts; that is, they treat the operand strictly as a bit pattern, without regard to sign. Consequently, if a signed number is shifted right, the sign bit is displaced one bit position to the right (like every other bit) and its value is replaced with a 0. If a signed number is shifted left, its sign bit is displaced into the carry flag and its value is replaced by the value of bit 6 (byte) or bit 14 (word). Clearly, your program must deal with this problem somehow.

What *should* happen for a true arithmetic shift left is that the processor should perform the regular logical shift left (which is what ASL does), but set a flag if the sign bit changes. Overflow (V) is the most appropriate flag here, so in your program, enter an instruction sequence similar to the one shown in Example 4-5 (assuming you’re shifting the accumulator).

An arithmetic shift right should preserve the sign of the operand by replicating the sign in the vacated high-bit position. Example 4-6 shows an instruction sequence that does this.

Example 4-5

; This routine shifts the accumulator left one bit position
; and sets the overflow (V) flag if the sign bit has changed.

```

                CLV                ;Clear overflow flag to start
                ASL  A              ;Shift the accumulator left
                BCC  POSITIVE
                BPL  OVERFLOW      ;C = 1. If N = 0, set overflow
                BMI  DONE
POSITIVE        BPL  OVERFLOW      ;C = 0. If N = 1, set overflow
OVERFLOW        SEP  #$40         ;Set the overflow flag
DONE            BVS  ERROR        ;Print an error message if V is 1

```

Example 4-6

; This routine shifts the accumulator right one bit position
; and preserves the sign.

```

                CMP  #0            ;Read sign of operand
                BPL  POSITIVE
                LSR  A              ;Operand is negative. Shift it,
                AND  #$8000        ; then set the sign bit
                BRA  DONE
POSITIVE        LSR  A              ;Operand is positive. Shift it
DONE            ..
                ..

```


Mode Control Instruction

XCE (Exchange Carry and Emulation Bits) is used to switch the 65816 between native mode and emulation mode. *XCE* takes no operand, but it uses the state of the carry flag to determine which mode to switch to. Specifically, $C = 1$ makes it switch to emulation mode, because *E* is 1 after the exchange; $C = 0$ makes it switch to native mode, because *E* is 0 after the exchange. Hence, switching to emulation mode requires

```
SEC
XCE
```

while switching to native mode requires:

```
CLC
XCE
```

XCE affects only the carry flag. And *because* it affects only *C*, switching to native mode causes the *M* and *X* bits to retain whatever settings bits 4 and 5 had in emulation mode. Recall that in emulation mode, bit 4 is the Break (*B*) bit and bit 5 is an unused bit that's always 1. Therefore, the *CLC/XCE* combination makes the accumulator 8 bits long (because bit 5 is 1) and the index registers either 8 or 16 bits long, depending on whether *B* was 1 or 0. Clearly, you can't leave all this to chance; you must set the *M* and *X* bits for the register lengths you want.

To switch to full native mode — i.e., with all registers 16 bits long — you must follow the *XCE* with a *REP* (Reset Processor Status Bits) instruction that clears *M* and *X* to 0. You must also tell the assembler to assume 16-bit memory and registers, using *LONGA ON* and *LONGI ON* directives. Therefore, the statements that switch the 65816 to full native mode are:

```
CLC                ;Switch to native mode
XCE
REP    #%110000    ;Make registers 16 bits long
LONGA  ON           ; and notify the assembler
LONGI  ON
```

When switching to emulation mode, you needn't set the register length, because registers are always 8 bits long. However, you must still tell the assembler what length you're using. The statements that switch the 65816 to emulation mode are:

```

SEC           ;Switch to emulation mode
XCE
LONGA  OFF    ;Tell the assembler that registers are
LONGI  OFF    ; 8 bits long

```

Interrupt-Related Instructions

Like a subroutine call, an interrupt from an external device makes the 65816 save return information on the stack, then transfer to an instruction sequence elsewhere in memory. However, a subroutine call makes the 816 execute a subroutine, while an interrupt makes it execute an *interrupt service routine*.

While the operands for the subroutine call instructions, JSR and JSL, can only be an absolute, absolute indexed indirect, or absolute long address, an interrupt always makes the 816 obtain the address of the service routine indirectly. That is, the 816 obtains the address from an *interrupt vector*, a 2-byte location at the end of bank 0.

The 65816 provides for 14 interrupt vectors, of which it uses 10 (see Table 4-15). Note that some interrupts are activated by signal lines on the microprocessor chip, while others are activated by the COP and BRK instructions.

There is yet another difference between subroutine calls and interrupts: a subroutine call saves only a return address on the stack, while an interrupt

Table 4-15

Location	Interrupt	Mode
00FFE4,5	COP instruction	Native
00FFE6,7	BRK instruction	Native
00FFE8,9	ABORT line	Native
00FFEA,B	NMI line	Native
00FFEC,D	-	-
00FFEE,F	IRQ line	Native
00FFF0,1	-	-
00FFF2,3	-	-
00FFF4,5	COP instruction	Emulation
00FFF6,7	-	-
00FFF8,9	ABORT line	Emulation
00FFFA,B	NMI line	Emulation
00FFFC,D	RESET line	Emulation and native
00FFFE,F	BRK installation and IRQ line	Emulation

saves the status flags as well. Specifically, when the 816 performs an interrupt, it does the following:

1. In native mode, pushes the program bank register (PBR) onto the stack.
2. Pushes the program counter's high byte, then its low byte, onto the stack.
3. Pushes the processor status register (P) onto the stack.
4. Sets the decimal mode (D) bit to 0, to specify binary mode.
5. Sets the IRQ disable (I) bit to 1, to lock out other interrupts while this one is being serviced.
6. Reads the contents of the interrupt vector into the program counter.

In summary, the PBR (in native mode), PC, and P registers are on the stack; D is 0 and I is 1, and the PC is pointing to the first instruction in the interrupt. That's where the 816 begins executing.

The 65816 has six interrupt-related instructions. Table 4-16 divides them into four groups: interrupt control, return from interrupt, software interrupts, and wait for interrupt.

Interrupt Control

These two instructions determine whether the 65816 accepts interrupt requests from external devices. *CLI* (*Clear Interrupt Disable Bit*) sets the IRQ disable bit (I) to 0, which lets the 816 respond to interrupt requests. *SEI* (*Set Interrupt Disable Bit*) sets the I bit to 1, which makes the 816 ignore requests. You generally disable interrupts when you're doing some time-critical or high-priority task that cannot be interrupted.

Return from Interrupt

RTI (*Return from Interrupt*) is to interrupts what *RTS* is to subroutines. That is, it undoes the work of the original operation and makes the 816 return to the main program. For this reason, *RTI* must be the last instruction the 816 executes in an interrupt service routine.

In emulation mode, *RTI* pulls the processor status register and the program counter off the stack. In native mode, it pulls P, PC, and the program bank register.

Table 4-16

		Flags							
Mnemonic	Assembler Format	N	V	M	X	D	I	Z	C
<i>Interrupt Control</i>									
CLI	CLI	0	.	.
SEI	SEI	1	.	.
<i>Return from Interrupt</i>									
RTI	RTI	*	*	*	*	*	*	*	*
<i>Software Interrupts</i>									
BRK	BRK or BRK dd	0	1	.	.
COP	COP	0	1	.	.
<i>Wait for Interrupt</i>									
WAI	WAI

Notes: (1) * means changed and . means unchanged.
 (2) Shaded instructions are new with the 65816.

Software Interrupts

BRK (Force Break) makes the 65816 act as it does when it responds to an external interrupt, except *BRK* also sets the program bank register to 0. As Table 4-15 shows, *BRK* has two interrupt vectors, one for each mode. In native mode, the 816 loads the contents of locations \$00FFE6 and \$00FFE7 into the PC's low and high bytes, respectively. In emulation mode, it loads the contents of \$00FFFE and \$00FFFF into the PC. In either case, executing a *BRK* instruction sends the Apple IIGS into its Monitor.

Although *BRK* is a 1-byte instruction, it makes the 816 increment the program counter by 2, thereby allowing you to insert a 1-byte identifier that indicates what condition caused the break; that is, *BRK* has two formats: *BRK* and *BRK nn*.

COP (Coprocessor) is similar to *BRK*, except that the 816 loads the PC from locations \$00FFE4 and 5 (native mode) or \$00FFF4 and 5 (emulation mode). Unlike *BRK*, *COP* is a true 2-byte instruction; the identifier operand is required. If you omit it, the assembler generates a "Missing Operand" error.

Wait for Interrupt

The final interrupt instruction, *WAI (Wait for Interrupt)*, does what its name says: it puts the processor into an idle state in which it halts and waits for an

interrupt or a hardware reset. (I suppose we all enter an idle state occasionally.) Making the 816 wait with WAI rather than with an endless loop has two advantages: (1) the idle state reduces power consumption and (2) it results in the quickest response to an interrupt.

Miscellaneous Instructions

Table 4-17 shows two instructions that don't fit in any other category. *NOP* (*No Operation*) is the simplest, because it does nothing whatsoever. That is, it affects no flags, registers (other than the program counter), or memory locations.

Surprisingly, *NOP* has a variety of uses. For example, it is convenient when you're developing programs. Suppose you have completed a portion of the program that includes a jump or branch to as yet unwritten code; *NOP* makes a handy target for the jump. You can also use *NOP*'s opcode (\$EA) to "patch" an object code file when you want to delete an instruction without reassembling the program.

The final instruction is (appropriately) *STP* (*Stop the Clock*). *STP* stops the processor and its clock, to reduce power consumption. The 816 does not restart until it receives a hardware reset.

There is also a noninstruction, the reserved mnemonic *WDM*. *WDM* will become an instruction when Western Design Center introduces its 32-bit microprocessor, the 65C832. Rumor has it that *WDM* stands for William D. Mensch, Jr., the 65816's designer!

Table 4-17

Mnemonic	Assembler Format	Flags							
		N	V	M	X	D	I	Z	C
NOP	NOP
STP	STP

Notes: (1) . means changed.
 (2) Shaded instruction is new with the 65816.

CHAPTER 5

Macros

A macro is a subroutinelike “miniprogram” that you can insert in a source program by mentioning its name. This chapter tells how to create macros and use them in programs. Developing a macro can be either a simple or complex task, depending on what you want the macro to do. Still, it’s quite possible that you will *never* find it necessary to develop macros of your own, because the *Apple IIGS Programmer’s Workshop (APW)* disk already contains a wide variety of them.

Most of these macros are actually “tool calls” — calls to subroutines in the Apple IIGS *Toolbox* (described in the next chapter). However, the *APW* disk also has a variety of macros that can be handy for doing general-purpose programming jobs. This chapter summarizes the most useful ones and describes how to use them in your programs.

Introduction to Macros

As just mentioned, a macro is a sequence of assembler statements (instructions and directives) that may appear several times in a program. Like subroutines, macros have names. Once you have defined a macro, you can enter its name in a source program anywhere you would normally enter the instruction sequence.

Macros Vs. Subroutines

Although macros and subroutines both provide a shorthand reference to a frequently used instruction sequence, they are not the same. The statements in a subroutine appear once in a program, and the processor transfers to them (or *calls* the subroutine) as needed. By contrast, the statements in a macro may occur many times within a program; the assembler replaces each mention of a macro name with the statements that name represents. (In computer terminology, the assembler “expands” the macro.) Therefore, when you execute the program, the processor executes the macro instructions “in-line”; it does not transfer elsewhere in memory, as it does with a subroutine. Hence, *a macro name is a user-defined assembler directive*; it issues commands to the assembler, rather than to the microprocessor.

Macros have two advantages over subroutines:

1. Macros are *dynamic*. You can easily change the way a macro operates (not merely what it operates on) each time, by changing its input parameters. By contrast, you can only vary the data that gets passed to a subroutine, making subroutines much more inflexible.
2. Macros make for faster-executing programs, because the processor is not delayed by call (JSR) and return (RTS) instructions, and the stack operations they employ, as it is with subroutines.

Nothing comes for free, however. Since a macro gets expanded every time it is used, it tends to make machine-language programs longer by filling memory with repeated instruction sequences. This is a drawback that subroutines do not have.

Macros Speed Up Programming

Like subroutines, macros can speed up your programming and debugging work, as well as any program updating you might do in the future. They speed up programming in that you create a macro just once, then use it wherever you want it in a program. Instead of entering a long sequence of instructions, you enter only the macro name that represents the sequence.

Macros can speed up debugging because you create and debug each macro individually. Once a macro is working properly, you never need to worry about whether *that* portion of your program is correct. You can concentrate on finding errors elsewhere.

Moreover, programs that include macros are generally easier to read and understand. Consequently, they are also easier to update and change.

Change the macro definition and the assembler automatically uses the new version everywhere it previously used the old one.

To see how macros can ease your workload, consider the instructions you need to display a character on the screen. (These instructions are for earlier Apple IIs. I will discuss their equivalents for the II GS later.) Displaying a character involves loading it into the accumulator, then calling the Monitor's COUT (Character Output) subroutine. To display a *D*, for example, requires:

```
LDA    #'D'        ;Select D for display
JSR    ($36)       ;Display it by calling COUT
```

Suppose you have a program that displays different letters from time to time. What does this involve on your part? It involves entering (and remembering) those same two instructions each time. Although there are only two instructions here, it's annoying to have to reenter them every time you need them. However, if you have defined the sequence as a macro called Cout, you can enter one of the following instead:

```
Cout D    ;Display D
Cout E    ;Display E
Cout y    ;Display y
```

Contents of Macros

Every macro definition has four parts:

1. A *MACRO* directive in the mnemonic (op code) field. This marks the beginning of a macro definition.
2. The *macro definition statement*, which has the following general form:

```
[&LAB] macro-name [&parm1[,&parm2[, . . .]]]
```

Note that in the definition only the macro name (in the mnemonic field) is required. The label specifier, &LAB is always optional. However, you should include it in every macro definition, to allow programs to jump or branch to that line.

The operand field lists any input parameters for the macro. These are the parameters you can change each time you call the macro. Each parameter name must begin with an ampersand (&)

symbol. Parameter names are separated with commas. For example, the macro definition statement for a macro that adds two values and stores the result in memory might look like this:

```
&LAB ADD &TERM1,&TERM2,&SUM
```

3. The *body* of the macro; the sequence of statements (instructions and directives) that define what the macro does.
4. An *MEND* directive in the mnemonic field, to mark the end of the macro definition.

For example, the following is a simple macro that adds two word-size values:

```
MACRO
&LAB ADDW    &TERM1,&TERM2,&SUM
      LONGA   ON
&LAB CLC
      LDA     &TERM1
      ADC     &TERM2
      STA     &SUM
      MEND
```

The assembler doesn't care whether you specify memory locations or immediate values for the operands (you can't use an immediate value for the sum, of course). As long as the final form is legal, the assembler makes the substitutions without complaining.

For example, at one place in the program you could add two memory locations by entering:

```
ADDW PRICE,TAX,COST
```

This makes the assembler insert the following instructions in the program:

```
CLC
LDA PRICE
ADC TAX
STA COST
```

Somewhere else, you could add an immediate value to a memory location by entering:

```
LB1  ADDW  MEMLOC, #4, MEMLOC
```

This time, the assembler would insert

```
LB1  CLC
      LDA  MEMLOC
      ADC  #4
      STA  MEMLOC
```

Note that this macro call is labeled LB1. Because the macro definition has a symbolic label, &LAB, on the CLC instruction, the assembler places the label you specify on the CLC instruction when it expands the macro.

These examples demonstrate how much easier it is to pass parameters to macros than to subroutines. With a macro, you enter the parameters on the same line as the macro's name; with a subroutine, you must put them in registers or memory locations.

Macro Directives

Table 5-1 summarizes the macro directives provided by the *Apple IIGS Programmer's Workshop*. They are divided into five groups: macro language, library, symbolic parameter, branching, and listing. (You can use the branching directives in source programs as well as macro definitions, but they are most useful in macros.)

Macro Language Directives

The MACRO and MEND directives have already been discussed; they simply mark the beginning and end of a macro definition. The MEXIT directive makes the assembler stop expanding the macro early. In effect, MEXIT does the same thing as MEND, except it ends the expansion somewhere within the macro definition, rather than at the end. You need MEXIT if your expansion can take several different paths, depending on the value of a variable. Path "decision" are controlled by the AIF conditional branch directive, which is described under "Branching Directives" later in this chapter.

Table 5-1

Directive	Function
Macro Language	
MACRO	Format: MACRO MACRO marks the beginning of a macro definition.
MEND	Format: MEND MEND marks the end of a macro definition.
MEXIT	Format: MEXIT MEXIT terminates a macro expansion, usually as the result of a branching directive.
Library	
MCOPY	Format: MCOPY <i>filename</i> MCOPY enters the specified <i>filename</i> in the list of available macro libraries. Once a file is in this list, the source program can use any macro in it. Up to four macro libraries can be active at any given time.
MDROP	Format: MDROP <i>filename</i> MDROP removes the specified <i>filename</i> from the list of available macro libraries.
MLOAD	Format: MLOAD <i>filename</i> MLOAD enters the specified <i>filename</i> in the list of available macro libraries, if it is not already there.
Symbolic Parameter	
LCLA	Format: LCLA <i>sparm</i> LCLA (Local Arithmetic) declares an arithmetic type symbolic parameter local to the current macro.
LCLB	Format: LCLB <i>sparm</i> LCLB (Local Boolean) declares a boolean type symbolic parameter local to the current macro.
LCLC	Format: LCLC <i>sparm</i> LCLC (Local Character) declares a character type symbolic parameter local to the current macro.
GBLA	Format: GBLA <i>sparm</i> GBLA (Global Arithmetic) declares an arithmetic type symbolic parameter global for the entire subroutine.
GBLB	Format: GBLB <i>sparm</i> GBLB (Global Boolean) declares a boolean type symbolic parameter global for the entire subroutine.
GBLC	Format: GBLC <i>sparm</i> GBLC (Global Character) declares a character type symbolic parameter global for the entire subroutine.

Table 5-1 (cont.)

Directive	Function
Symbolic Parameter (cont.)	
SETA	Format: <i>sparm</i> SETA <i>aexp</i> SETA (Set Arithmetic) resolves the arithmetic expression in the operand field to a four-byte signed hexadecimal number and assigns it to the symbolic parameter in the label field.
SETB	Format: <i>sparm</i> SETB <i>bexp</i> SETB (Set Boolean) evaluates the boolean expression in the operand field as either true or false, and assigns either 1 or 0 (respectively) to the symbolic parameter in the label field.
SETC	Format: <i>sparm</i> SETC <i>cexp</i> SETC (Set Character) evaluates the expression in the operand field as a character string and assigns it to the symbolic parameter in the label field.
AMID	Format: <i>sparm</i> AMID <i>string</i> , <i>start-pos</i> , <i>#-chars</i> AMID extracts a substring from a specified <i>string</i> and assigns it to the symbolic parameter in the label field. The substring begins at the character position numbered <i>start-pos</i> (the first character in the string is at position 1) and is <i>#-chars</i> characters long. Note that AMID does the same thing as the MID\$ function in BASIC.
ASEARCH	Format: <i>sparm</i> ASEARCH <i>target\$</i> , <i>search\$</i> , <i>start-pos</i> ASEARCH searches the string <i>target\$</i> , starting at character position <i>start-pos</i> , for the first occurrence of the <i>search\$</i> substring. If the substring is found, its starting position is assigned to the symbolic parameter in the label field; otherwise, the parameter is set to zero.
Branching	
AGO	Format: AGO <i>ssym</i> AGO makes processing continue with the statement that follows the specified sequence symbol (see "Branching Directives").
AIF	Format: AIF <i>bexp</i> , <i>ssym</i> AIF evaluates the Boolean expression <i>bexp</i> . If the expression is true, processing continues with the statement that follows the specified sequence symbol (see "Branching Directives"); if false, processing continues with the statement that follows the AIF directive.
Listing	
GEN	Format: GEN ON/OFF GEN ON causes all lines generated by macro expansions to be included on the assembler's output listing. GEN OFF lists only macro definitions on the output listing.
TRACE	Format: TRACE ON/OFF The assembly normally omits conditional assembly directives from its output listing. TRACE ON makes it list these lines.

Library Directives

The *MCOPY* directive enters the name of a macro library file into a list of available macro libraries. This makes the macros in the library available to the source program. For example,

```
MCOPY NEW.MACROS
```

activates the macro library file called NEW.MACROS.

Up to four macro library files can be active at any given time. Thus, your source program may include up to four *MCOPY* directives. A more detailed discussion of macro libraries is upcoming, in the “Creating Macro Libraries” section.

The *MDROP* directive removes a specified file from the list of available macro libraries. You only need *MDROP* if you are juggling more than four libraries.

MLOAD is similar to *MCOPY*, except *MLOAD* enters a file name into the list of available macro libraries only if the name is not already in the list.

Symbolic Parameter Directives

These directives operate on symbolic parameters — variables within a macro definition.

The first three directives — *LCLA*, *LCLB*, and *LCLC* — tell the assembler that a symbolic parameter is internal, or “local,” to a particular macro expansion; it is unknown throughout the rest of the program. The directives *GBLA*, *GBLB*, and *GBLC* tell the assembler that a symbolic parameter is “global”; its name can be referred to anywhere within the program.

The *SETA*, *SETB*, and *SETC* directives assign the value of an expression to a symbolic parameter. Thus, these directives do for symbolic parameters what the *EQU* (Equate) directive does for regular symbols.

The *AMID* directive extracts a substring from a string and assigns the substring to a symbolic parameter. (Think of *AMID* as the assembler counterpart of BASIC’s *MID\$* function.) *AMID* takes three operands: the string, the character position where the substring begins (position 1 is the first character), and the length of the substring. For example,

```
&SUBS  AMID  'THE STRING', 5, 3
```

assigns the substring STR to &SUBS.

Finally, *ASEARCH* searches a string for a specified substring. If the substring is found, the symbolic parameter receives the position of its first character; otherwise, the parameter is set to 0.

Branching Directives

The assembler provides two branching directives, one conditional (*AIF*) and the other unconditional (*AGO*). *AIF* is the assembler's counterpart of the microprocessor's conditional branch instructions (BEQ, BMI, and so on), while *AGO* is the assembler's BRA. Just as the microprocessor's branch instructions can make the microprocessor continue executing at an instruction label elsewhere in memory, the branching directives can make the assembler continue processing at a *sequence symbol* elsewhere in a macro definition. A sequence symbol is a label that begins with a period (.) and is on a line by itself.

AIF has the general form

```
AIF bexp, ssym
```

where *bexp* is a boolean (logical) expression and *ssym* is the target sequence symbol. If *bexp* is true, the assembler continues processing at *ssym*; otherwise, if *bexp* is false, the assembler continues processing with the statement that follows *AIF*.

AGO makes the assembler continue processing a specified sequence symbol unconditionally. It's generally used to skip past a block of statements to which a preceding "true" *AIF* branches.

To see how *AIF* and *AGO* might be used, consider the *ADDW* addition macro described earlier. *ADDW* takes three parameters, the two terms to be added and the sum. Its macro definition line is:

```
&LAB ADDW &TERM1, &TERM2, &SUM
```

Now suppose we want to make the *&SUM* term optional. In this case, the sum is stored in *&SUM* if the user supplies it; otherwise, if he or she omits *&SUM*, the sum is stored in *&TERM1*. To make this happen, the macro definition would contain the following kinds of statements:

```
. .
. .
.C
    AIF    C:&SUM>0, .D    ;Is &SUM an empty string?
```

```

        LCLC  &SUM          ; Yes. Declare it local
&SUM   SETC  &TERM1        ; and set it to &TERM1
        AGO   .E            ; Skip .D and continue
                                ; at .E

.D
        . . .              ; Process these instruc-
                                ; tions if the user
        . . .              ; entered an &SUM term

.E
        . . .              ; Do the addition
        . . .
        . . .

```

Listing Directives

The assembler normally shows only macro call statements in its output listing; it does not “expand” macros to show their contents. You can, however, make it list expansions by entering a *GEN ON* directive. A subsequent *GEN OFF* will turn off the expansion listing.

Similarly, the assembler normally omits conditional assembly directives from the output listing. You can make it include them by entering a *TRACE ON* directive.

Creating Macro Libraries

As mentioned earlier, a macro library is a text file that contains one or more macro definitions. To use the macros in a library, your source program must make the library active by reading it with an *MCOPY* directive.

There are two ways to create a macro library:

1. If all the macros are new, you can enter their definitions using the editor, then save the file with the name you want to use in your *MCOPY* directive.

To keep things simple, you should give the library the same name as the program that will use it, and end the name with *.MACROS*; for example, *SORT.MACROS* is a reasonable name for a file that contains macros used by a program called *SORT*.

2. To use macros that are contained in one or more existing library files, you can use the *MACGEN* utility contained on the *Programmer's Workshop* disk.

Here's a description of method 2. The MACGEN (Macro Library Generator) utility scans an assembler source file for macro names. To start it, enter a command of the form:

```
MACGEN [ +C/-C ] source-file out-file macro-lib1
          [macro-lib2 . . .]
```

MACGEN's parameters are as follows:

- +C (the default) removes all excess blanks from macro definitions; specify -C if you have used GEN ON or TRACE ON in your source file.
- *source-file* is the name of your source file.
- *out-file* is the name of the macro library file you want to generate — the file name that appears in the program's MCOPY directive.
- *macro-lib1*, *macro-lib2*, and so on, are the files to be searched for the macro names that appear in *source-file*. Note that the filenames are separated by spaces.

This command makes MACGEN scan *source-file*, and any files named in its COPY and APPEND directives, for macro names mentioned in the program. It then opens a temporary file called SYSMAC on the active disk and reads the macro definitions from the *macro-lib* files into it. When MACGEN has resolved all macros, it changes the name of SYSMAC to *out-file* — the filename in your program's MCOPY directive. For example,

```
MACGEN MYPROG.SRC MYPROG.MACROS FILE1.MACROS
      FILE2.MACROS
```

searches FILE1.MACROS, then FILE2.MACROS, for macro definitions whose names are mentioned in MYPROG.SRC, and copies those definitions into a new file called MYPROG.MACROS.

You can also use MACGEN to update a macro library file if you add new macro calls to your program. Simply specify the file that contains the new macro definitions as *macro-lib1* and the existing library file as *macro-lib2* and . For example,

```
MACGEN MYPROG.SRC MYPROG.MACROS FILE3.MACROS
      MYPROG.MACROS
```


scans MYPROG.SRC and updates MYPROG.MACROS by adding macro definitions contained in FILE3.MACROS.

You may be tempted to bypass MACGEN, and MCOPY the relevant libraries into your source file directly. There are two reasons why you should resist that temptation:

1. The assembler reads library files *very* slowly, so copying large libraries or multiple libraries takes much longer than copying a smaller file that MACGEN has tailored to your source program.
2. Some macros call other macros. MACGEN will seek out every needed macro definition and put it in your output file. You may overlook some of these internally called macros, and wind up with assembly errors.

Macros on the *Programmer's Workshop* Disk

The *Apple IIGS Programmer's Workshop* (APW) disk has a subdirectory called LIBRARIES/AINCLUDE that contains macro files for each tool set in the IIGS Toolbox (described in Chapter 6). However, it also has two files that are unrelated to the Toolbox. M16.PRODOS contains macros that let your programs perform ProDOS 16 commands, while M16.UTILITY provides general-purpose macros for performing both 8- and 16-bit operations.

ProDOS 16 Macros

The ProDOS 16 macros, listed in Table 5-2, are macro versions of the function calls in ProDOS's Machine Language Interface (MLI). Using them saves you from having to remember the numeric "opcode" for each call (shown in *italics* in the table); each macro supplies it automatically.

Note that each macro name is preceded by an underscore character (). The *Programmer's Workshop* uses the underscore prefix to identify macros that employ system calls of any kind.

Utility Macros

The macros in file /APW/LIBRARIES/AINCLUDE/M16.UTILITY perform a variety of fundamental operations that are useful for developing

assembly language programs. As Table 5-3 shows, these macros fall into nine groups: push and pull, load and store, add and subtract, define, move, shift, mode, write, and check error. Except for *native*, all utility macros assume the processor is operating in full (16-bit) native mode.

These macros are generally self-explanatory, so I don't provide a formal description. However, if a macro requires one or more operands, I give examples of its various forms. The examples should illustrate clearly what the macro does.

Pay special attention to the *pushword* and *pushlong* macros. Most calls to tools in the Apple IIGS Toolbox require input values or pointers to those values to be on the stack. Hence, you will usually precede a tool call with one or more *pushword* (for 2-byte inputs) or *pushlong* (for 4-byte inputs) calls.

The *str* (string) macro is also useful for defining messages to be displayed on the screen.

Push and Pull Macros

push1 [opr] Push 1 byte onto the stack

```

pushl    Loc                ; Push byte at Loc
pushl    Loc,x              ; Push byte at Loc,x
pushl    #n                 ; Push constant n
pushl    ;                  ; Push byte from A

```

pushword [opr] Push 2 bytes onto the stack

push2	Loc	; Push bytes at Loc
push2	Loc, x	; Push bytes at Loc, x
push2	#n	; Push constant n
push2		; Push bytes from A

push3	opr[,reg]	Push 3 bytes onto the stack
--------------	-----------	-----------------------------

push3	Loc	;Push bytes at Loc
push3	Loc,x	;Push bytes at Loc,x
push3	#n	;Push constant n

pushlong	addr[,offset]	Push 4 bytes onto the stack
-----------------	---------------	-----------------------------

`pushlong Loc ; Push bytes at Loc`

Table 5-2

Format	Description
__ALLOCINTERRUPT DCB	Install an interrupt handler \$40
__CHANGEPATH DCB	Change a file's pathname \$04
__CLEARBACKUPBIT DCB	Clear the backup bit in the file's access byte \$0B
__CLOSE DCB	End access to file \$CC
__CREATE DCB	Create file or directory \$C0
__DEALLOCINTERRUPT DCB	Remove an interrupt handler \$41
__DESTROY DCB	Delete file or directory \$C1
__FORMAT DCB	Format a disk \$24
__FLUSH DCB	Empty file's I/O buffer \$CD
__GETBOOTVOL DCB	Get name of volume from which PRODOS file was last executed \$28
__GETDEVNUM DCB	Get device number \$20
__GETEOF DCB	Get size of file in bytes \$D1
__GETFILEINFO DCB	Get file information \$C4
__GETLASTDEV DCB	Get number of the last device accessed \$21
__GETLEVEL DCB	Get system file level \$1B
__GETMARK DCB	Get current position in file \$CF
__GETVERSION DCB	Get ProDOS 16 version number \$2A
__GETPATHNAME DCB	Get the current application's pathname \$27
__GETPREFIX DCB	Get current path name prefix \$C7
__NEWLINE DCB	Enable new line read mode \$C9
__OPEN DCB	Prepare file for access \$C8
__QUIT DCB	Terminate the current application \$29
__READ DCB	Read bytes from file \$CA
__READBLOCK DCB	Read 512 bytes from file \$80
__SETEOF DCB	Set size of file \$D0
__SETFILEINFO DCB	Set file information \$C3
__SETLEVEL DCB	Set system file level \$1A
__SETMARK DCB	Set new position in file \$CE
__SETPREFIX DCB	Set pathname prefix \$C6
__VOLUME DCB	Get the disk's name, its size in blocks, the number of free (unallocated) blocks, and the file system identification number \$08
__WRITE DCB	Write bytes to file \$CB
__WRITEBLOCK DCB	Write 512 bytes to file \$81

```

pushlong  #Loc           ; Push address of Loc
pushlong  Loc, x         ; Push bytes at Loc, x
pushlong  #n             ; Push constant n
pushlong  [zeropg], offset ; Push using indirect
                           ; addressing

```

pushxy

Push 4 bytes onto the stack from X and Y

Table 5-3

Format	Description
Push and Pull Macros	
push1 [opr]	Push 1 byte onto the stack
pushword [opr]	Push 2 bytes onto the stack
push3 addr[,reg]	Push 3 bytes onto the stack
pushlong addr[,offset]	Push 4 bytes onto the stack
pushxy	Push 4 bytes onto the stack from X and Y
pushay	Push 4 bytes onto the stack from A and Y
pull1 [opr]	Pull 1 byte from the stack
pullword [opr]	Pull 2 bytes from the stack
pull3 addr	Pull 3 bytes from the stack
pulllong [addr1[,addr2]]	Pull 4 bytes from the stack
pullxy [addr]	Pull 4 bytes from the stack using X and Y
pullay	Pull 4 bytes from the stack into A and Y
pullx [addr]	Pull 2 bytes from the stack using X
Load and Store Macros	
lday addr[,offset]	Load A and Y (4 bytes)
stay addr[,offset]	Store A and Y (4 bytes)
zero bytes,addr	Zero a block
Add and Subtract Macros	
add [term1],term2[,result]	Add 2-byte integers
add4 [term1],term2[,result]	Add 4-byte integers
sub [term1],term2[,result]	Subtract 2-byte integers
sub4 [term1],term2[,result]	Subtract 4-byte integers
Define Macros	
str 'string'	Define string
dp pointer	Define pointer
Move Macros	
move1 from,to[,to2]	Move 1 byte
moveword from,to[,to2]	Move 2 bytes
move3 from,to[,to2]	Move 3 bytes
movelong from,to[,to2]	Move 4 bytes
Shift Macros	
asl4 addr[,count]	Left-shift 4 bytes
lsr4 addr[,count]	Right-shift 4 bytes
Mode Macros	
native [long/short]	Turn on native mode
emulation	Turn on emulation mode
long	Set memory, A, X, and Y to 16 bits
longm	Set memory and A register to 16 bits

Table 5-3 (cont.)

Format	Description
Mode Macros (cont.)	
longx	Set X and Y registers to 16 bits
short	Set memory and registers to 8 bits
shortm	Set memory and A register to 8 bits
shortx	Set X and Y registers to 8 bits
Write Macros	
writex [opr]	Write a character
writestr [opr]	Write a string
writeln [opr]	Write a line (string + CR)
Check Error Macro	
Check__Error error__subr	Call error subroutine if carry is 1

pushay Push 4 bytes onto the stack from A and Y

pull1 [opr] Pull 1 byte from the stack

```

pull1 Loc          ; Push byte and store
                    ; it at Loc
pull1 Loc,x        ; Push byte and store
                    ; it at Loc,x
pull1              ; Push byte into A

```

pullword [opr] Pull 2 bytes from the stack

```

pull12 Loc         ; Pull bytes and store
                    ; them at Loc
pull12 Loc,x       ; Pull bytes and store
                    ; them at Loc,x
pull12             ; Pull bytes into A

```

pull3 addr Pull 3 bytes from the stack

```

pull13 Loc         ; Pull bytes and store
                    ; them at Loc

```

pulllong [addr1[,addr2]] Pull 4 bytes from the stack

```

pulllong  Loc                ; Pull bytes and store
                                ; them at Loc
pushlong  Loc1, Loc2         ; Pull bytes and store
                                ; them at both Loc1
                                ; and Loc2
pulllong  [zeropg], offset   ; Pull bytes and store
                                ; them using indirect
                                ; addressing
pulllong                                     ; Pull 4 bytes. Discard
                                ; first 3, store
                                ; second 2 in A

```

pullxy [addr] Pull 4 bytes from the stack using X and Y

```

pullxy                ; Pull into X and Y
pullxy Loc            ; Pull into X and Y,
                        ; store at Loc

```

pullay Pull 4 bytes from the stack into A and Y

pullx [addr] Pull 2 bytes from the stack using X

```

pullx                ; Pull into X
pullx Loc            ; Pull into X, store
                        ; also at Loc

```

Load and Store Macros

lday addr[,offset] Load A and Y (4 bytes)

```

lday  Loc                ; Load A from Loc, Y
                                ; into Loc+
lday  #n                 ; Load A and Y with
                                ; constant n
lday  [zeropg], offset   ; Load A and Y
                                ; indirectly
lday  zp, x              ; Load A from zp, x, Y
                                ; from zp+2, x

```

stay	addr[,offset]	Store A and Y (4 bytes)
-------------	----------------------	--------------------------------

stay	Loc	; Store A into Loc, Y
		; into Loc+2
stay	[zeropg], offset	; Store A and Y
		; indirectly
stay	zp, x	; Store A into zp, x, Y
		; into zp+2, x

zero	bytes,addr	Zero a block
-------------	------------	--------------

zero	#n, block	; Clear the first <i>n</i>
		; bytes of block
zero	#n, [Loc]	; Clear the first <i>n</i>
		; bytes of the block
		; whose address is in
		; <i>Loc</i>
zero	num, block	; Obtain the byte count
		; from memory

Add and Subtract Macros

add	[term1],term2[,result]	Add 2-byte integers
------------	------------------------	---------------------

```

add    Loc1,Loc2,Loc3      ;Add Loc1 to Loc2 and
                           ; store the sum in
                           ; Loc3
add    Loc1,Loc2           ;Add Loc1 to Loc2 and
                           ; return the sum in A
add    ,Loc2,Loc3          ;Add Loc2 to A and
                           ; store the sum in
                           ; Loc3
add    #4,Loc2             ;Add 4 to Loc2 and
                           ; return the sum in A

```

add4	[term1],term2[,result]	Add 4-byte integers
	See <i>add</i> for examples.	

sub	[term1],term2[,result]	Subtract 2-byte integers
------------	------------------------	--------------------------

```

sub      Loc1,Loc2,Loc3      ;Subtract Loc2 from
                             ; Loc1 and store the
                             ; result in Loc3
sub      Loc1,Loc2           ;Subtract Loc2 from
                             ; Loc1 and return the
                             ; result in A
sub      ,Loc2,Loc3          ;Subtract Loc2 from A
                             ; and store the
                             ; result in Loc3
sub      Loc1,#4             ;Subtract 4 from Loc1
                             ; and return the
                             ; result in A

```

sub4 [term1],term2[,result] Subtract 4-byte integers
 See *sub* for examples.

Define Macros

str 'string' Define string
 Generates a Pascal-type string — that is, a 1-byte character count followed by the string. For example,

```
ThisString str 'This is my string'
```

produces the statement:

```
ThisString dc il'17',c'This is my string'
```

dp pointer Define pointer
 Calculates the 4-byte address of the operand and puts it in a DC statement of the form:

```
dc i4'pointer'
```

Move Macros

move1 from,to[,to2] Move 1 byte

```

move1    Here,There          ;Copy the byte at Here
                             ; into There
move1    Here,There,There2    ;Copy the byte at Here

```



```

; into both There
; and There2
movel    #n, There      ; Copy the constant n
; into There

```

moveword from,to[,to2] Move 2 bytes

```

moveword  Here, There      ; Copy the bytes at
; Here into There
moveword  Here, There, There2 ; Copy the bytes at
; Here into both
; There and There2
moveword  #n, There        ; Copy the constant n
; into There
moveword  [zeropg], offset, ; Copy 2 bytes to There
; using There in-
; direct addressing

```

move3 from,to[,to2] Move 3 bytes

```

move3    Here, There      ; Copy the 3 bytes at
; Here into There
move3    Here, There, There2 ; Copy the 3 bytes at
; Here into both
; There and There2
move3    #n, There        ; Copy the constant n
; into There

```

movelong from,to[,to2] Move 4 bytes

```

movelong  Here, There      ; Copy the bytes at
; Here into There
movelong  Here, There, There2 ; Copy the bytes at
; Here into both
; There and There2
movelong  #n, There        ; Copy the constant n
; into There
movelong  [zeropg], offset, ; Copy 4 bytes to There
; using There in-
; direct addressing

```

```

movelong Here,x,There      ;Copy the bytes at
                             ; Here plus index X
                             ; into There
movelong Here,y,There      ;Copy the bytes at
                             ; Here plus index Y
                             ; into There

```

Important: For the last two formats, *x* and *y* must be in lowercase.

Shift Macros

asl4 addr[,count] Left-shift 4 bytes

```

asl4   Loc,#3               ;Left-shift contents
                             ; of Loc by 3 bit
                             ; positions
asl4   Loc,CountLoc         ;Left-shift contents
                             ; of Loc by count
                             ; stored in CountLoc
asl4   Loc                  ;Left-shift contents
                             ; of Loc by count
                             ; stored in X
                             ; register

```

lsr4 addr[,count] Right-shift 4 bytes

```

lsr4   Loc,#3               ;Right-shift contents of
                             ; Loc by 3 bit positions
lsr4   Loc,CountLoc         ;Right-shift contents of
                             ; Loc by count stored in
                             ; CountLoc
lsr4   Loc                  ;Right-shift contents of
                             ; Loc by count stored in X
                             ; register

```

Mode Macros

native [long/short] Turn on native mode

```

native long      ;Native mode with 16-bit registers
native           ;Same as preceding
native short     ;Native mode with 8-bit registers

```

emulation	Turn on emulation mode
long	Set memory, A, X, and Y to 16 bits
longm	Set memory and A register to 16 bits
longx	Set X and Y registers to 16 bits
short	Set memory, A, X, and Y to 8 bits
shortm	Set memory and A register to 8 bits
shortx	Set X and Y registers to 8 bits

Write Macros

writetch [opr] Write a character

```

writetch                ;Write character in A
                        ; register
writetch    #'A'        ;Write an "A"
writetch    Loc,x       ;Write character
                        ; addressed by Loc,x

```

writestr [opr] Write a string

```

writestr                ;A and Y point to string
writestr    #'Press a key' ;Write "Press a key"
writestr    Loc          ;Write string start-
                        ; ing at Loc

```

writeln [opr] Write a line (string + CR)

```

writeln                ;Write Carriage Return
                        ; only
writeln    #'Press a key' ;Write "Press a key",
                        ; then return
writeln    Loc           ;Write line starting
                        ; at Loc

```

Check Error Macro

Check__Error error__subr Call error subroutine if carry is 1
 The user-defined subroutine *error__subr* can be anywhere in the current program bank.

Using Predefined Macros

The *APW* disk's LIBRARIES/AINCLUDE subdirectory contains more than 20 macro library files. To use a macro in your program, you must copy its definition into your program's macro file using MACGEN.

Eventually, you will know instinctively where specific macros are located. Until then, however, the easiest way to copy the definitions your program needs is to make MACGEN search the *entire* directory, by giving it the equal sign (=) wildcard. Hence, your MACGEN command would be of the form:

```
macgen myprog.src myprog.macros /apw/libraries/  
ainclude/ml6. =
```

CHAPTER 6

The Apple IIGS Toolbox

In Chapter 2, I introduced the Apple IIGS Toolbox, the large collection of functionally-based *tool sets*. I also mentioned that tool sets are comprised of assembly language routines called *tools*. Some tool sets are built into ROM, others are entirely on the System Disk, and a few are divided between ROM and the System Disk.

To use tools that are on the System Disk, you must first read them into the computer's memory. Apple's manuals refer to such tools as *RAM-based tools*, to distinguish them from tools that are built into ROM. In time, if Apple adds more ROM to the IIGS, it's likely that all the tools will be stored in ROM. Until then, your program must have some way of knowing where to obtain the tools it needs.

Fortunately, the ROM includes a tool set called (appropriately) the *Tool Locator* that finds tools for you. You must use the Tool Locator in every program that includes tool calls, and that will be virtually every program you write for the IIGS. Two other tool sets that get involved in most programs are the Memory Manager and QuickDraw II, so I'll discuss all the tool sets briefly, but these three I'll describe in detail. After that, I will demonstrate how to initialize tool sets and how to access the tools within them. Finally, I will present some guidelines for using tools and provide a program "model" that includes the instructions, assembler directives, and tool calls that programs generally need.

Important: Like most software products, the Apple IIGS Programmer's Workshop will probably change from time to time. Apple may add, delete, and modify tools — or even tool sets — and thereby affect the accuracy of this book. For example, the name of a tool may change. Although I have no control over that, I intend to keep the book as up-to-date as possible. To do that, I ask for your help. If you encounter something that appears to be inaccurate in the book, please write to me in care of the publisher.

Tool Locator

When your program requests use of a tool (by issuing a tool “call”), the Tool Locator looks up the memory address of that tool and gives it to the 65816. Looking up the address takes two steps, and they involve numbers that the Apple IIGS designers have assigned to tool sets and the tools within them.

First, the Tool Locator uses the tool set number to look up the 4-byte address, or *pointer*, in a tool set address table. This pointer points to a second table, one that holds the addresses of the individual tools in the tool set. The Tool Locator uses the tool number to obtain from this second table the address of the routine for that tool — and that is the execution address it gives to the microprocessor.

A picture is worth 76 words here. Figure 6-1 shows how a tool call (SetPenMode, in this case) makes the Tool Locator look up the two pointers and eventually direct the 816 to the tool's routine in memory.

Memory Manager

The Memory Manager is a ROM-based tool set that controls the use of memory in the Apple IIGS; think of it as the computer's bookkeeper. When you load a program from disk, the System Loader calls the Memory Manager to request space for it. The Memory Manager, in turn, allocates a block of memory for the program and tells the System Loader the block's address.

A program always remains where the System Loader puts it; it is at a *fixed* location in memory. Not all blocks in memory are fixed, however; blocks that programs set up to hold data are often *relocatable*. The Memory Manager is free to move them if it needs that space for a program or another data block. For this reason, when your program requests a block of memory, the Memory Manager has a noteworthy way of reporting where that block is located.

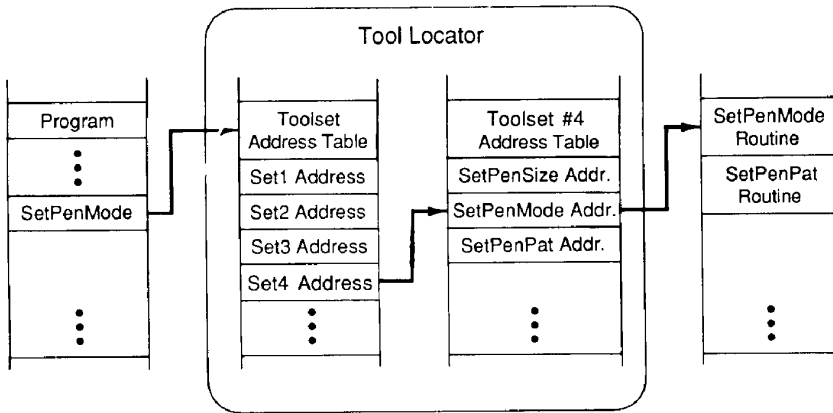


Figure 6-1

In response to your program's request, the Memory Manager allocates the block and puts its address in a 4-byte location called the *master pointer*. It does not give this address to the program, because it may move the block. Instead, the Memory Manager gives the program the address of the master pointer. This address is called the block's *handle*.

Note the logic behind this technique: the block may move (and its address change), but the address of the master pointer (i.e., the handle) never changes. The handle is, then, an identification number that the program can use at any time to access the allocated block, regardless of where it is in memory.

Even if your application needs no memory blocks for its own use, some of the Toolbox's tool sets require working space in memory bank 0, and your program must ask the Memory Manager to allocate it. (For example, QuickDraw II requires three pages, or 768 bytes). Finally, at the end of your program, you must ask the Memory Manager to free up or "deallocate" your memory block. To do this, you tell it to discard the block's handle.

QuickDraw II

Most, if not all, of your programs will involve displaying something on the screen. If the screen is in one of the regular Apple II graphics or text display modes, you could use Apple II emulation to produce your picture or text. However, to take advantage of the new super high-resolution mode (the

standard output for 65816 native-mode programming), you will need the services of the Apple IIGS's video display tool set, QuickDraw II.

QuickDraw II is responsible for every kind of display operation: drawing windows, graphics shapes (rectangles, ovals, arcs, etc.), menus, and text characters. Yes, even text! With super high-resolution, text is displayed as bit-mapped graphics, not as the built-in text characters the other display modes use. I'll describe QuickDraw II in more detail, and show you how to use it, in the next chapter.

Managers

In addition to the Tool Locator, Memory Manager, and QuickDraw II — which virtually all programs use — the Apple IIGS Toolbox contains a variety of other tool sets. Some have names ending with “Manager,” because they manage a certain type of operation.

Window Manager

The Window Manager lets you display windows. The simplest window is the blank full screen, but you can also create more complex windows that have a menu bar, scroll bars, and other components. The Window Manager also keeps overlapping windows under control.

The Window Manager relies on QuickDraw II to draw the elements in a window's border and the picture or text within it, and on the Control Manager (described shortly) to manage its components.

Menu Manager

The Menu Manager helps you create Macintosh-style menu bars and menus that you “pull down” from it using the mouse. Figure 6-2 shows an Edit menu that has been pulled down from a menu bar.

When you pull down a menu, the Menu Manager keeps track of which item you're pointing to. As soon as you release the mouse button, the Menu Manager reports the selection to the application program. The program, in turn, is responsible for executing whatever instructions are associated with that item.

The Menu Manager relies heavily on QuickDraw II's drawing and color capabilities. Since menus are part of a window, it also relies on the Window Manager.

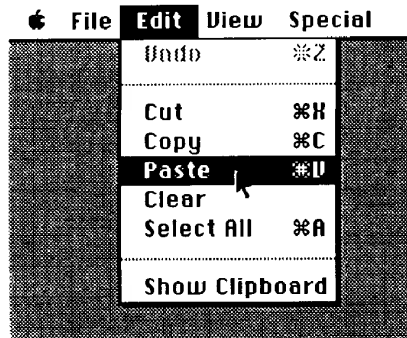


Figure 6-2

Control Manager

The Control Manager regulates on-screen *controls* — graphics objects that you activate with the mouse. The following controls are defined within the Control Manager:

- *Buttons* cause an immediate action when clicked or pressed with the mouse. They appear on the screen as round-cornered rectangles with a title centered inside.
- *Check boxes* act like on/off switches; they are checked with an X when on and empty (unchecked) when off. Check boxes are used to choose options from a list. They are generally used to specify some future action, instead of causing an immediate action of their own.
- *Radio buttons* are also on/off controls. But while any number of check boxes may be on, clicking a radio button on turns off all other radio buttons in that group. That is, they act like the station-selector buttons on a car radio.
- *Window scroll bars* are dials in which the moving part, called the “scroll bar” (or, sometimes, “thumb”), shows the position of the window relative to the total size of the document or picture being displayed.

Figure 6-3 shows the predefined controls, plus two other kinds of dials you

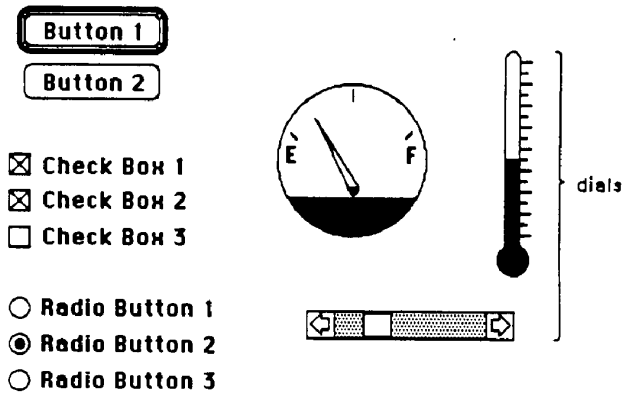


Figure 6-3

can define using the Control Manager. Since controls are graphics objects, the Control Manager works in conjunction with QuickDraw II; and because the controls are placed in a window, it also relies on the Window Manager.

Event Manager

Most programs interact with the user, letting him or her make selections or issue commands by pressing keys or clicking the mouse. The Apple IIGS manuals refer to these actions — pressing a key or clicking the mouse — as *events*. Some interactive programs sit and wait for an event from the user. This is common in educational programs, for example, where the program must wait for the user to answer a question. Other interactive programs, such as games, don't wait for events; they simply continue running, and deal with events as they occur.

The Event Manager keeps track of events by continually monitoring the keyboard and mouse. Every time the user presses a key or clicks the mouse button, the Event Manager takes note of what happened and records it in a chronological log called the *event queue*. (A queue is a stack that operates in “first-in/first-out” fashion, like a vending machine that dispenses candy. Compare this with the 65816's stack, which operates in “last-in/first-out” fashion, like a machine that holds trays in a cafeteria.)

The application program must, then, examine the event queue periodically to see if it contains any event records. If there are any records in the queue, the program must extract each one and do what it signifies (if anything), then continue processing records until the queue is empty.

To draw an analogy, the Event Manager acts like a receptionist in an office. Since the boss (the application program) is always busy, the Event Manager answers telephone calls (key and mouse button presses) and notes them on “While You Were Out” forms (event records). Each form is put on the boss’s desk (the event queue), beneath any that are already there. From that point, it’s up to the boss to respond.

Dialog Manager

Commands in menus normally perform one operation. (For example, “Quit” normally terminates the program.) However, commands that need more information from the user must present a *dialog box* (a rectangle that may contain text, controls, and icons) to request that information. Figure 6-4 shows a dialog box that lets the user specify the page setup for a print operation. Dialog boxes are the responsibility of the Dialog Manager.

Desk Manager

The Desk Manager handles small, memory-resident programs that usually provide on-screen equivalents of desk accessories — calculators, calendars, clocks, notepads, and so on. Being in memory, a desk accessory program can be summoned while a main application program is running; that is, the

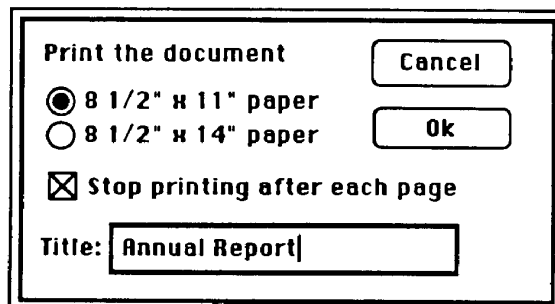


Figure 6-4

user can summon the desk accessory and do something with it, then continue the application as if nothing had happened. If you want to be able to use desk accessory programs while your application is running, you must activate the Desk Manager.

The Apple IIGS can handle two kinds of desk accessories, called (perhaps with a tip of the hat to Coca-Cola) *classic* and *new*. Classic desk accessories (CDAs) can be summoned from either 6502 emulation mode or 65816 native mode. To get a list of the installed CDAs, you press OpenApple-Control-Esc — the key combination that makes the Apple IIGS Control Panel available. Indeed, the Control Panel is itself a classic desk accessory. The Control Panel is built into ROM; other CDAs must be provided on disk.

New desk accessories (NDAs) can only be called from native mode. If the application program has provided for NDAs, their names appear automatically when the user selects the Apple menu on the menu bar at the top of the screen. With these names on the screen, you can activate any NDA by moving the mouse pointer to it and releasing the button. You may already be familiar with NDAs; they're the style of desk accessories available on the Macintosh.

Sound Manager

The *Sound Manager* uses the Ensoniq Digital Oscillator Chip (DOC) in the Apple IIGS to produce tones through the speaker. The DOC contains 32 individual oscillators, and you can combine them in pairs to produce up to 15 “voices.” A related tool set called the *Note Synthesizer* works with the Sound Manager and the DOC to create a polyphonic music synthesizer.

Other Managers

There are also a few other managers worth mentioning. The *Print Manager* lets your program print text or graphics without concern as to what kind of printer is connected to the computer.

The *Font Manager* lets you display text in any of several styles or “fonts.” It also lets you create fonts of your own design.

The *List Manager* lets you operate on lists of information, to search them, sort them, and so on.

The *Scrap Manager* provides the tools you need to move or copy text or graphics between programs (either two application programs, an application program and a desk accessory, or two desk accessories). The Scrap Manager gets its name from the fact that it keeps the data being transferred in a block of memory called the *desk scrap*.

Other Tool Sets

In addition to the managers, there are six other major tool sets: the Line Editor, Text Tools, Integer Math Tools, Standard File Operations, SANE, and miscellaneous tools.

Line Editor

The Line Editor (or *LineEdit*) provides basic line-editing capabilities. Among the things it lets you do are:

- Insert new text.
- Delete characters by backspacing over them.
- Select text to be cut or deleted using the arrow keys, or dragging through it with the mouse.
- Move or copy blocks of text.

Text Tools

There are two ways to display text on a text-mode screen. One way is to run your program in emulation mode in bank 0 and call the Monitor subroutines that output text. A better way is to use the Apple IIGS Text Tools. The Text Tools output text with the 65816 running in native mode in any bank.

Integer Math Tools

The Integer Math Tools include routines for operating on 2-byte and 4-byte signed integers, and on signed fixed-point numbers and their fractional parts. The operations the Integer Math Tools can do include multiplication, division, square root, sine, cosine, arc tangent, rounding, and conversions between data types.

Standard File Operations

The Standard File Operations tool set allows you to open and store files from within an application program. It can work with files on any drive in the system.

SANE

SANE, short for Standard Apple Numerics Environment, is the tool set for doing high-precision fixed-point and floating-point math operations. For

fixed-point computations, it uses a 64-bit data type. For floating-point, it can use any of three data types: single (32-bit), double (64-bit), or extended (80-bit). Besides rudimentary operations such as add, subtract, multiply, divide, and square root, SANE can do such jobs as:

- Compare numbers.
- Produce logs, exponentials, and trigonometric functions.
- Convert between binary and decimal or floating-point and integer.
- Calculate compound interest and annuity functions for financial institutions.
- Generate random numbers.

Miscellaneous Tools

With the tool calls in the miscellaneous tool set, you can:

- Access the battery backed-up RAM that keeps track of the date and time. This area of memory includes all the Control Panel parameters, including date and time, display mode and colors (foreground, background, and border), and the bell.
- Read the current time and date for display by your program. You can also use the current time value to calculate elapsed time or generate a waiting period or delay.
- Transfer data to or from peripheral cards.
- Call some of the Monitor subroutines from full native mode.

Tool Set Interactions

As mentioned earlier in this chapter, some tool sets involve other tool sets in their work. Recall, for example, that the Menu Manager relies on QuickDraw II to draw menus. Similarly, the Window Manager relies on QuickDraw II to draw its border elements and on the Control Manager to manage its border elements. Although the way tool sets interact may appear to be a complex and puzzling web, it will become clearer once you have gained some programming experience.

While the tool sets can't be ranked in order of importance, they can be

ranked in order of *involvement* — according to how other tool sets depend on them. Figure 6-5 is an attempt to show the interdependencies of the tool sets. Here, the sets toward the top depend on the ones below them. You can, if you wish, use this figure as a guide for your programming. As a rule of thumb, for a given tool set to work properly, you should have activated all the tool sets shown below it.

Using Tool Calls in Programs

Tool calls are generally macro calls, so you create a program that makes tool calls just as you create one that calls any of the general-purpose macros described in Chapter 5. That is:

1. Create the source program using the editor. At the beginning of the program, enter an *MCOPY lib-file* directive, where *lib-file* is the name of the macro library file that will contain tool call macros and other macros used in the program.
2. Enter each tool name at the place in the program where you want to use it.

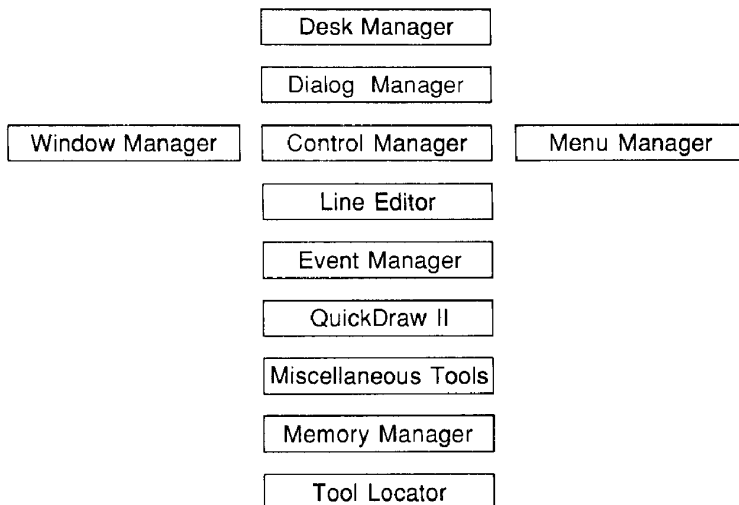


Figure 6-5

3. Save the program on disk.
4. The tool call macros are in the /APW/MACROS directory on the Programmer's Workshop disk. Therefore, to construct the macro library, enter a command of the form:

```
MACGEN source-file lib-file /APW/MACROS/=
```

5. Assemble and link your program with

```
ASML source-file
```

if the program contains a KEEP directive, or else with:

```
ASML source-file KEEP=out-file
```

Making Tool Calls

There are hundreds of tool calls, and they are all described in the two-volume *Apple IIGS Toolbox Reference*. Apple's software designers have given the tools simple, one-word names to make them easy to understand and remember. For example, the calls `WindStartup` and `WindShutDown` activate and deactivate the Window Manager. Other calls to the Window Manager include `NewWindow`, `CloseWindow`, `SetWTitle`, `SetFrameColor`, `ShowWindow`, `HideWindow`, and `MoveWindow`.

That's how the tool names appear in the *Toolbox Reference*. In assembly language programs, you must precede each tool name with an underscore (`__`) character. For example, you must enter `WindStartup` as `__WindStartup`, `WindShutDown` as `__WindShutDown`, and so on.

Calling Conventions

Some tools (`WindShutDown`, for example) require no input values and produce no result value, but these are the exception rather than the rule. Most tools require input values or produce a result, or both. Tools that pass values to and from the application use the *stack* to do so.

This means that before you call a tool that requires input values, you must push those values onto the stack. It also means that before you call a tool that produces a result value, you must reserve space for it on the stack. Finally, if a tool both requires inputs and produces a result, you must do

both operations: reserve space for the result, *then* push the inputs onto the stack. Hence, calling a tool can take up to four steps:

1. Push space on the stack for the result (if any).
2. Push the input parameters onto the stack.
3. Make the tool call.
4. Pull the result (if any) from the stack.

Inputs and the result are usually 2- or 4-byte values, so the standard way to reserve stack space or push data is by using the M16.UTILITY macros *PushWord* and *PushLong*. For example, you would use the following kind of sequence to call a tool that requires three inputs (2 words and a long word, in this case) but does not return a result:

```
PushWord    param1      ; Push a 2-byte value,
PushWord    param2      ; then another
PushLong    param3      ; Push a 4-byte value
_ToolName
```

Conversely, the following sequence calls a tool that returns a 2-byte result, but requires no input values:

```
PushWord    #0          ; Reserve space for a 2-byte
                        ; result
_ToolName
```

Here, *PushWord* pushes a 0, but any value would do. After all, you're simply reserving space; the value you push is never used.

Finally, this sequence calls a tool that takes three inputs and returns a 2-byte result:

```
PushWord    #0
PushWord    param1
PushWord    param2
PushLong    param3
_ToolName
```

Again, because the stack operates in first-in/last-out fashion, the push that reserves space for the result must *always* come first. After the tool routine

has removed the inputs from the stack, it is left with space in which to insert the result.

Words, Integers, and Pointers

Tool calls work with two kinds of 2-byte values; unsigned values are called *words*, while signed values are called *integers*. The most common kind of 4-byte value (or longword) is a memory address or *pointer*. Keep those terms in mind as you read the rest of this book.

General Structure of an Application Program

I have no way of knowing the details of the programs you want to create, but every Apple IIGS program that uses the Toolbox must have the same general elements. Specifically, every program must:

1. MCOPY the macro file for your program.
2. Set the data bank equal to the program bank, so you can use absolute addressing to access any data that may be in your program.
3. Start the Tool Locator, Memory Manager, and Miscellaneous Tools. Miscellaneous Tools and every other tool set makes the carry flag 1 if it cannot be started for some reason. Therefore, you must exit immediately if a Startup call returns $C = 1$.
4. Some tool sets need working space in bank 0. Request it from the Memory Manager.
5. Start QuickDraw II and any other ROM-based tool sets your application uses.
6. Read any needed RAM-based tools from disk.
7. Start the RAM-based tools.
8. Run your application.
9. Shut down all the tool sets you used.

Using the Program Bank as the Data Bank

Setting the data bank equal to the program bank takes only two instructions:

```

phk          ; Push the program bank register onto the
              ; stack,
plb          ; then pull it into the data bank register

```

Starting the Tool Locator and Memory Manager

To start the Tool Locator and Memory Manager, enter:

```

_TLStartup
PushWord #0  ; Reserve space for result (ID number)
_MMStartup

```

The Memory Manager returns a word-size program identification number that you must use to request bank 0 space for the tools that need it. Save this number with:

```

pla          ; Pull program ID from the stack
sta  MyID    ; and store it in memory

```

Start Miscellaneous Tools with a sequence of the form:

```

_MTStartup
ldx #3
jsr PrepareToDie

```

The *PrepareToDie* subroutine should check the carry flag, and exit the program with an error message if carry is 1. Here, the X register receives the error code (3, in this case) that is to be included in the error message.

Allocating Working Space in Bank 0

The tool sets listed in Table 6-1 need working space in bank 0. Note that QuickDraw II needs three pages (where a page is 256 or \$100 bytes), while the rest need only one page. To obtain this space from the Memory Manager, you must issue a *NewHandle* call.

NewHandle takes four inputs from the stack: a longword that specifies how many bytes you need, a word that contains your program ID (the one you stored in *MyID*), a word that specifies the type of memory block being requested (fixed or relocatable), and a longword that specifies the desired

Table 6-1

Tool Set	Bytes Required
QuickDraw II	\$300
Event Manager	\$100*
Control Manager	\$100**
Menu Manager	\$100
Line Editor	\$100
Font Manager	\$100
Print Manager	\$100
Sound Manager	\$100
Standard File Operations	\$100
SANE	\$100

*The Event Manager shares its page with the Window Manager.

**The Control Manager shares its page with the Dialog Manager.

location of the block (if any). NewHandle returns a longword *handle* on the stack, so you must reserve stack space for it. (Recall from the “Memory Manager” section that a handle is a pointer to a pointer. In this case, the handle points to the address of the memory block.) For example, to request seven pages in bank 0, enter:

```

PushLong    #0          ; Reserve stack space for result
              ; (handle)
PushLong    $$700       ; Request 7 pages
PushWord    MyID        ; Program ID
PushWord    $$C005      ; Locked, fixed location, fixed
              ; bank
PushLong    #0          ; Location
_NewHandle

```

After that, you must retrieve the handle from the stack and store it in the direct page (the 65816's zero page). QuickDraw and other tool sets also need to know the address of their direct page space, so you must get the pointer to it using indirect addressing. Instructions that do this are:

```

pla          ; Pull handle from stack
sta    0     ; and store it in direct page
pla
sta    2

lda    [0]   ; Get pointer to memory block
sta    4     ; and store it at loc. 4 of DP

```

Note that although the pointer is a 3-byte address, the *sta 4* instruction only stores the lower 2 bytes. It's unnecessary to store the bank number because the direct page is always in bank 0.

Starting ROM-Based Tool Sets

In addition to the Tool Locator, Memory Manager, and Miscellaneous Tools, nine other tool sets — including QuickDraw II and the Event Manager — are built into ROM. (I'll provide the full list shortly.) Just as you start the Tool Locator, Memory Manager, and Miscellaneous Tools using `__TLStartup`, `__MMStartup`, and `__MTStartup`, you start every other tool set using a call that ends with "Startup." For example, `__QDStartup` starts QuickDraw II, `__EMStartup` starts the Event Manager, and so on.

I describe each Startup call as I discuss its tool set throughout the rest of the book. Each Startup call sets the carry flag to 1 if an error occurred. Thus, you should follow the Startup with a *jsr PrepareToDie* sequence, as I did when starting Miscellaneous Tools.

Reading RAM-Based Tools from Disk

The RAM-based tool sets are in the System Disk's /SYSTEM/TOOLS sub-directory, and they are loaded into RAM when you boot up. To use any RAM-based tool set in a program, you must tell the computer which ones you want by calling LoadTools. The disk also contains additional tool calls for some of the ROM-based tool sets; to use these extra tools, you must specify their tool sets to LoadTools, too.

The generalized calling sequence for LoadTools is:

```
PushLong ToolTablePtr
_LoadTools
```

where *ToolTablePtr* points to a table of the form:

```
ToolTable    dc i'NumTools'
              dc i'ToolNum1, MinVersion'
              dc i'ToolNum2, MinVersion'
              .
              .
              dc i'ToolNumN, MinVersion'
TTEnd        anop

TableSize    equ  TTEnd-ToolTable-2
NumTable     equ  TableSize/4
```

Here, *NumTools* specifies the number of entries in the rest of the table. The equates at the end make the assembler calculate its value.

Each entry consists of the tool set's identification number and the number of the earliest or "minimum" version of that set with which your program can work. (More about minimum version numbers shortly.) Table 6-2 lists the identification numbers Apple has assigned to the tool sets and indicates where each resides, in ROM or on disk.

The Apple IIGS is an evolving computer, and Apple will issue updates to the tool sets that add new tool calls or perhaps replace or change existing tool calls. The minimum version (*MinVersion*) parameter in each tool table

Table 6-2

ROM-Based Tools	
1	Tool Locator
2	Memory Manager
3	Miscellaneous Tools
4	QuickDraw II
5	Desk Manager
6	Event Manager
7	Scheduler
8	Sound Manager
9	Apple Desktop Bus
10	SANE
11	Integer Math Tools
12	Text Tools
13	(Used internally)
RAM-Based Tools	
14	Window Manager
15	Menu Manager
16	Control Manager
17	Loader
18	High-Level Printer Driver
19	Low-Level Printer Driver
20	Line Editor
21	Dialog Manager
22	Scrap Manager
23	Standard File Operations
24	Disk Utilities
25	Note Synthesizer
26	Note Sequencer
27	Font Manager

entry lets your program adapt to changes within tool sets. That is, it lets you specify the earliest version of a tool set with which the program can work.

Tool set version numbers are of the form XX.xx, where XX and xx are the major and minor version numbers. Hence, a tool set whose version number is 1.02 has a major version number of 1 and a minor version number of 02. In the tool table, you must enter the minimum version number as a word value, where the upper byte gives the major version number and the lower byte gives the minor version number. For example, to specify version 1.02 as the minimum version number, enter \$0102 in the tool table.

Every tool set includes a "Version" call that returns the current version number (QuickDraw II's call is QDVersion, for example), and you would use them to obtain the number for the tool sets you're now using. However, if you don't care which version the user has, or you only intend to use the program on your own computer, you can enter \$0100 for each minimum version number. For example, the following LoadTools call activates eight RAM-based tool sets:

```
InitStuff START
    using GlobalData
    . .
    . .
    PushLong #ToolTable
    _LoadTools
    . .
    . .
    END

GlobalData DATA
    . .
    . .
    ToolTable dc i'NumTools'      ;No. of tool sets in
                                   ; the table
              dc i'4,$0100'       ;QuickDraw
              dc i'5,$0100'       ;Desk Manager
              dc i'6,$0100'       ;Event Manager
              dc i'14,$0100'      ;Window Manager
              dc i'15,$0100'      ;Menu Manager
              dc i'16,$0100'      ;Control Manager
              dc i'20,$0100'      ;Line Editor
              dc i'21,$0100'      ;Dialog Manager
```

```

TTend      anop

TableSize  equ  TTend-ToolTable-2
NumTools   equ  TableSize/4
.
.
.
END

```

Note that the LoadTools call is in a code segment (InitStuff), whereas the ToolTable is in a data segment (GlobalData). This is the proper way to separate instructions and data in a IIGS application program. Note also that InitStuff must declare that it is *using GlobalData*, so the assembler knows where to look for the ToolTable.

Of course, for LoadTools to work correctly, the tool sets it selects must be on disk. If they aren't, LoadTools sets the carry flag to 1 and loads a ProDOS error code into the accumulator. An error code of \$45 indicates "Volume directory not found," which means, in this case, that ProDOS couldn't find the tools in memory; any other code indicates some other ProDOS error.

If LoadTools returns carry equal to 1, there are several things your program can do. It can, of course, simply give up — that is, display a message and break using PrepareToDie. However, that's a rather harsh way to punish a user who simply booted with the wrong disk. It's more reasonable to tell the user to insert the disk, then try loading the tools again. If LoadTools still returns a carry of 1, the program should shut down the tools and exit.

Example 6-1 shows instructions that do all this (and assumes you want the eight tool sets I loaded in the preceding example). In fact, it displays not just a prompt, but a pseudo *dialog box*. The box contains the prompt and two buttons that let the user signal that he or she has either inserted the disk (OK) or wants to quit the program (Cancel).

This program fragment has three code segments (Progame, InitStuff, and MountBookDisk) and one data segment (GlobalData). The main segment, Progame, contains a call to an InitStuff subroutine and a branch instruction that exits if InitStuff cannot load the tools.

InitStuff calls LoadTools, which checks whether the tools listed in the ToolTable are in memory; if not, it attempts to load them from disk. If the tool sets can't be loaded (carry is 1), InitStuff checks the accumulator for the "Volume directory not found" code, \$45. This value makes the program branch to DoMount; any other makes it exit with PrepareToDie.

At DoMount, the program calls a MountBookDisk subroutine. To

Example 6-1

```

Progame START
    ..
    ..
    jsr  InitStuff      ;Initialize the tools
    bcs  AllDone        ;Quit if tools couldn't be initialized
    ..                ;Otherwise, continue here
    ..
    END
InitStuff START
    using GlobalData
    ..
    ..
LoadAgain PushLong  #ToolTable
    _LoadTools
    bcc  ToolsLoaded    ;Error?

    cmp  #VolNotFound   ; Yes.
    beq  DoMount        ;If it's $45, get user to insert disk
    sec                      ;Otherwise, exit
    ldx  #$FE
    jsr  PrepareToDie
DoMount  anop
    jsr  MountBootDisk  ;Get user to mount the disk
    cmp  #1             ;Has correct disk been mounted?
    beq  LoadAgain     ; Yes. Try loading again
    sec                      ; No. Set error flag
    rts                ; and return to caller
ToolsLoaded  anop      ;Tools have been loaded
    ..
    ..
    END
MountBootDisk START

    _Set_Prefix SetPrefixParams
    _Get_Prefix GetPrefixParams

    PushWord #0          ;Space for result
    PushWord #195        ;Column position for dialog box
    PushWord #30         ;Row position for dialog box
    PushLong #PromptStr  ;Prompt at top of dialog box
    PushLong #VolStr     ;Volume name string
    PushLong #OKStr      ;String in Button 1
    PushLong #CancelStr  ;String in Button 2
    _TLMountVolume
    pla                  ;Obtain the button number
    rts                  ; and return to caller
PromptStr str 'Please insert the disk.'
VolStr    ds 16
OKStr     str 'OK'
CancelStr str 'Shutdown'

GetPrefixParams dc i'7'
                dc i4'VolStr'

```

176 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

```
SetPrefixParams dc i'7'
                dc i4'BootStr'
BootStr    str '*/'
            END

GlobalData DATA

ToolTable dc i'NumTools' ;No. of tool sets in the table
          dc i'4,$0100' ;QuickDraw
          dc i'5,$0100' ;Desk Manager
          dc i'6,$0100' ;Event Manager
          dc i'14,$0100' ;Window Manager
          dc i'15,$0100' ;Menu Manager
          dc i'16,$0100' ;Control Manager
          dc i'20,$0100' ;Line Editor
          dc i'21,$0100' ;Dialog Manager
TTEnd      anop

TableSize equ TTEnd-ToolTable-2
NumTools equ TableSize/4

VolNotFound equ $45 ;ProDOS error
            END
```

begin, MountBookDisk calls SetPrefix and GetPrefix to read the current volume name into a 16-byte location called VolStr (Volume String). Then it calls TLMountVolume, a tool that displays a dialog box and waits for the user to “press” one of two buttons using the mouse. By pressing *OK* (button 1), the user indicates the System Disk has been inserted; by pressing *Shut-down* (button 2), he or she terminates the program.

Starting RAM-Based Tool Sets

Like ROM-based tool sets such as QuickDraw II, the RAM-based tool sets have calls that end in “Startup.” I will describe each Startup call as I discuss its tool set throughout the rest of the book.

Shutting Down the Tool Sets

Just as you must start up tool sets before you can use them, you must shut them down when you finish using them. Each tool set has a call ending with “ShutDown.” For example, __QDShutDown shuts down QuickDraw II, __MenuShutDown shuts down the Menu Manager, and so on. None of the ShutDown calls return a result value, and only the Memory Manager call, __MMShutDown, requires an input. To shut down the Memory Manager, you must push your program’s I.D. number (the number produced by __MMStartup) onto the stack. To do this, enter calls of the form:

```
PushWord MyID
__MMShutDown
```

I'll give the entire shut-down sequence later.

Leaving the Program

To leave the program and return to the Program Launcher or the *Programmer's Workshop*, you must give a ProDOS Quit command. The tool call that does this, Quit, has the general form:

```
_Quit QuitParams
```

where *QuitParams* is defined by:

```
QuitParams  dc  i4'0'
             dc  i'$4000'
```

Here, the four-byte 0 value tells ProDOS to return to the calling program (APW shell or Program Launcher), while the word-size \$4000 value tells it to keep the program in memory. Having a program in memory allows you to run it later without reloading it from disk.

Quit can produce four different errors (“Path not found”, “File not found”, “Out of memory”, or “Not an executable system file”) that leave the computer in an unknown state. For this reason, you should follow the Quit call with a *brk \$F0* instruction to force an exit.

Generalized Program Model

In the preceding section, I listed and described the elements that every program must have. I will now pull all of these elements together in a generalized program *model* that includes the “boilerplate” most programs need. Once you have created the model file, you can use it as a starting point to produce every program you write.

Example 6-2 shows the model. Although it's rather long, realize that (fortunately) you must only enter it once.

The model contains many tool calls that I have not yet described. Please bear with me — if I described everything now, you would probably become both utterly confused and totally bored. Look over the listing, but don't try to understand every single detail. Rest assured that I will describe these calls eventually, at places where I use them throughout the rest of the book.


```

ToolTable      dc i'NumTools'          ;No. of tool sets in table
               dc i'4,$0100'           ;QuickDraw
               dc i'5,$0100'           ;Desk Manager
               dc i'6,$0100'           ;Event Manager
               dc i'14,$0100'          ;Window Manager
               dc i'15,$0100'          ;Menu Manager
               dc i'16,$0100'          ;Control Manager
               dc i'20,$0100'          ;LineEdit
               dc i'21,$0100'          ;Dialog Manager
TTEnd          anop

TableSize      equ TTEnd-ToolTable-2
NumTools       equ TableSize/4

VolNotFound    equ $45                ;ProDOS error

END

```

```

;*****
;
; InitStuff
;
; Initializes tool sets, gets space in bank 0 for use as zero
; page by tool sets that need it, and ensures that RAM-based
; tools are in memory.
;
;*****

```

```

InitStuff      START
               using GlobalData

```

```

; Initialize the Tool Locator, Memory Manager, and Miscellaneous
; Tools.

```

```

      _TLStartup          ;Tool locator

      PushWord #0          ;Memory Manager
      _MMStartup

      pla                 ;Memory Manager returns program's ID
      sta MyID

      _MTStartup          ;Misc. Tools
      ldx #3
      jsr PrepareToDie

```

```

; Get some space for the direct page we need. QuickDraw needs
; three pages and the Event Manager, Control Manager, Line
; Editor, and Menu Manager each need one page.

```

```

      PushLong #0          ;Space for handle
      PushLong #$700       ;Seven pages
      PushWord MyID        ;Owner
      PushWord #$C005      ;Locked, fixed, fixed bank
      PushLong #0          ;Location
      _NewHandle
      ldx #$FF
      jsr PrepareToDie

      pla                 ;Read handle and store in dp
      sta 0

```

180 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

```

        pla
        sta 2

        lda [0]                ;Get dp location from handle
        sta 4                  ; and store at loc 4 of dp

; Initialize QuickDraw

        pha                    ;Use dp obtained from handle
        PushWord #ScreenMode   ;Mode = 640
        PushWord #160          ;Max size of scan line (in bytes)
        PushWord MyID
        _QDStartup
        ldx #4
        jsr PrepareToDie

; Initialize Event Manager

        lda 4                  ;dp to use = QD dp + $300
        clc
        adc #$300
        sta 4
        pha
        PushWord #20           ;Queue size
        PushWord #0            ;X clamp left
        PushWord #MaxX         ;X clamp right
        PushWord #0            ;Y clamp top
        PushWord #200          ;Y clamp bottom
        PushWord MyID
        _EMStartup
        ldx #6
        jsr PrepareToDie

;-----
;
; Load the RAM-based tools I need.

LoadAgain    PushLong #ToolTable
              _LoadTools
              bcc ToolsLoaded

              cmp #VolNotFound
              beq DoMount
              sec
              ldx #$FE
              jsr PrepareToDie

DoMount      anop
              jsr MountBootDisk
              cmp #1
              beq LoadAgain

              sec
              rts

; The tools have been loaded. Initialize the Window Manager, Control
; Manager, LineEdit, Dialog Manager, Menu Manager, and Desk Manager.

ToolsLoaded  anop
              PushWord MyID    ;Window Manager
              _WindStartup

```

```

ldx #14
jsr PrepareToDie

PushLong #0                ;Prepare screen for windows
 RefreshDesktop

PushWord MyID              ;Control Manager
lda 4                      ;DP to use = EM DP + $100
clc
adc #$100
sta 4
pha
_CtlStartup
ldx #16
jsr PrepareToDie

PushWord MyID              ;LineEdit
lda 4
clc
adc #$100
sta 4
pha
_LEStartup
ldx #20
jsr PrepareToDie

PushWord MyID              ;Dialog Manager
 DialogStartup
ldx #21
jsr PrepareToDie

PushWord MyID              ;Menu Manager
lda 4
clc
adc #$100
pha
_MenuStartup
ldx #15
jsr PrepareToDie

_DeskStartup              ;Desk Manager

clc                        ;Clear the carry flag
rts                        ; and return

END

;*****
;
; Event Loop
;
; This is the subroutine that interacts with the user.
;
;*****

EventLoop      START
               using GlobalData
               rts
               END

```

182 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

```
*****
*
* Prepare To Die
*
* Dies if carry is set.
*
* Error number to display is in X register.
* Assumes that Miscellaneous Tools is active.
*
*****
```

```
PrepareToDie  START
               bcs RealDeath      ;Carry = 1?
               rts                 ; No. Return to caller

RealDeath     phx                 ; Yes. Goodbye, program.
               PushLong #DeathMsg
               _SysFailMgr

DeathMsg      str 'Could not handle error '

               END
```

```
*****
;
; Mount Boot Disk
;
; Tells the user to insert the disk that contains the RAM-based
; tools.
;
*****
```

```
MountBootDisk  START

               _Set_Prefix SetPrefixParams
               _Get_Prefix GetPrefixParams

               PushWord #0          ;Space for result
               PushWord #195        ;Column position for dialog box
               PushWord #30         ;Row position for dialog box
               PushLong #PromptStr  ;Prompt at top of dialog box
               PushLong #VolStr     ;Volume name string
               PushLong #OKStr      ;String in Button 1
               PushLong #CancelStr  ;String in Button 2
               _TLMountVolume

               pla                  ;Obtain the button number
               rts                 ; and return to caller

PromptStr      str 'Please insert the disk.'
VolStr         ds 16
OKStr          str 'OK'
CancelStr      str 'Shutdown'

GetPrefixParams dc i'7'
               dc i4'VolStr'
SetPrefixParams dc i'7'
               dc i4'BootStr'

BootStr        str '*/'

               END
```


What's In the Model

The model has one data segment (*GlobalData*) that contains the program's variables. There are also five code segments.

The *Model* segment contains the program's starting and ending points, as well as global equates that are used throughout the program. It contains a call to *InitStuff*, which starts the tool sets the program uses. From *MTStartup* on, each Startup sequence ends by calling *PrepareToDie* with the tool set number in the X register. If carry is 1 (the startup failed), *PrepareToDie* exits with an error message.

InitStuff also calls *LoadTools* to read the RAM-based tools from disk. If *LoadTools* is unsuccessful, a *MountBootDisk* subroutine prompts the user to insert the correct disk. If *InitStuff* still cannot load the tools, it sets the carry flag to 1 and returns to the Model segment, where a Branch on Carry Set (bcs) instruction closes down the program.

If all tools have been successfully initialized, the program calls an *EventLoop* subroutine (empty at the moment) to interact with the user. When the user quits, Model shuts down the tool sets and makes a ProDOS Quit call to return to the system program.

Creating the Model File

Create the model file now, using the editor, and name it MODEL.SRC (i.e., enter **edit model.src**). Before starting the editor, be sure to give an **asm65816** command so that MODEL.SRC will be an assembler source code file.

When you finish, print a copy of MODEL.SRC and compare it closely with Example 6-2. If you find any typographical errors or omissions, correct them using the editor.

Validating MODEL.SRC

To help ensure that you have entered the MODEL.SRC file correctly, you should assemble and link it. Before you can assemble, however, you must create the MODEL.MACROS file using MACGEN. To do this, enter the command:

```
macgen model.src model.macros /apw/libraries/  
ainclude/ml6.=
```

If you mistyped one or more tool calls, MACGEN displays their names and asks for the name of a file to search. That's your cue to press Esc and leave MACGEN so you can make the corrections.

If MACGEN runs without errors, assemble and link the model file by entering:

```
asml model.src keep=model`
```

The *keep* option is necessary here because the model does not contain a KEEP directive. (You need a KEEP in *one* of these places to generate the object and load files.)

Putting KEEP on the command line is handy because it lets you create multiple versions of a program. For example, suppose you have developed a program called MYPROG and it doesn't do quite what you want. You can load MYPROG.SRC into the editor, make the required changes, then save it as, say, MYPROG1.SRC. Now you have two different versions of the same program! When you finally get a version that works, delete all the others and rename the working version with the original name (MYPROG, in this case).

Providing that the assembler and linker report no errors, the preceding ASML command produces three new files: object (OBJ) files MODEL.ROOT and MODEL.A, and the shell load (EXE) file MODEL. These files are simply by-products of the verification procedure. They aren't needed, so you can delete them. However, retain MACGEN's output, MODEL.MACROS, because it's a handy macro library file for MACGENs you perform on behalf of your application programs.

Using the Model

To create an application program using MODEL.SRC, do the following:

1. With the *Workshop* prompt on the screen, enter **edit model.src** to start the editor with the model file.
2. Do a Search and Replace Down (OpenApple-J) command to replace *Model* with the name of your program.
3. Delete tool calls and instructions your program doesn't need. The editor's Cut (OpenApple-X) command is handy for deleting groups of lines.

If there is an entire tool set that you don't need, follow these rules:

- Delete the tool set's Startup sequence from the InitStuff segment and its ShutDown call from the Model segment.
 - If the tool set is RAM-based, delete its entry from the ToolTable list in the GlobalData segment.
 - If the tool set is one that requires space in bank 0, change the operand for the second PushLong in the calling sequence for __NewHandle (in the InitStuff segment).
4. If your program requires a tool set that is not in the model, do the following:
 - Insert its Startup and ShutDown sequences in the InitStuff and Model segments, respectively.
 - If the tool set is RAM-based, insert its entry in the ToolTable list in the GlobalData segment. For the tool set number, see Table 6-2 under "Reading RAM-Based Tools from Disk."
 - The Sound Manager, Standard File Operations, and SANE each require a page in bank 0. To use one of these tool sets, you must add \$100 to the operand for the second PushLong in the __NewHandle calling sequence (this is in the InitStuff segment).
 5. Insert the data, instructions, and tool calls for your program.
 6. Leave the editor and select the N (Save to a new name) option to save the file with the name you want. Then select the E (Exit without updating) option to make the APW prompt reappear.
 7. To generate the macro library file for your program, enter a command of the form:

```
macgen myprog.src myprog.macros model.macros
/apw/libraries/ainclude/ml6. =
```

where MODEL.MACROS is the macro library file produced by the MACGEN you performed to verify MODEL.SRC. (If you deleted this library, omit its name from your MACGEN command. MACGEN will then create MYPROG.MACROS from scratch.)

Copying Between Programs

You needn't start every new program using the model, of course; you may want to construct a new program based on a similar program that already exists. The editor's copy and paste (OA-C and OA-V) commands are handy for copying code sequences between programs.

As you may recall, the editor always stores copied text *on disk*, which means that it remains intact when you switch programs — or turn the computer off, for that matter. Thus, to reproduce an existing instruction sequence in a new program, simply highlight it (with OA-C and the arrow keys), save it (Return), exit (Ctrl-Q), load the new program (L), and paste the copy where you want it (OA-V).

Tool Locator, Memory Manager, and Miscellaneous Tools Calls

The model program in this chapter contains many tool calls that I have not yet covered. As promised, I will discuss them later. However, I *have* described some calls to the Tool Locator, Memory Manager, and Miscellaneous Tools; Table 6-3 summarizes them.

Start-Up and Shut-Down Sequences

The Apple IIGS requires its major tool sets to be started in the following order:

- Tool Locator
- Memory Manager
- Miscellaneous Tools
- QuickDraw
- Event Manager
- Window Manager
- Control Manager
- Line Editor
- Dialog Manager
- Menu Manager
- Desk Manager

Font Manager, Standard File, and other tool sets not listed here.

Table 6-3

Tool Locator	
__TLStartup	Start the Tool Locator
Call with: __TLStartup	
Result: None	
__TLShutDown	Shut down the Tool Locator
Call with: __TLShutDown	
Result: None	
__LoadTools	Load RAM-based tools from boot disk
Call with: PushLong ToolTablePtr	;Pointer to tool table
__LoadTools	
Result: None	
Note: ToolTablePtr points to	
dc i'NumTools'	
dc i'ToolNum1,MinVersion'	
dc i'ToolNum2,MinVersion'	
..	
..	
dc i'ToolNumN, MinVersion'	
__LoadOneTool	Load specified tool set from boot disk
Call with: PushWord #ToolNum	;Tool set number
PushWord #MinVersion	;Minimum version number
__LoadOneTool	
__TLMountVolume	Display a dialog box with a prompt
Call with: PushWord #0	;Space for result
PushWord WhereX	;Column pos. for dialog box
PushWord WhereY	;Row pos. for dialog box
PushLong PromptStr	;Prompt at top of dialog box
PushLong VolStr	;Volume name string
PushLong But1Str	;String in Button 1
PushLong But2Str	;String in Button 2
__TLMountVolume	
Result: Button number (word)	

Memory Manager

__MMStartup	Start the Memory Manager
Call with: PushWord #0	;Space for result
__MMStartup	
Result: Program ID (word)	
__MMShutDown	Shut down the Memory Manager
Call with: PushWord ProgID	;Program ID
__MMShutDown	
Result: None	

Table 6-3 (cont.)

Memory Manager (cont.)			
<hr/>			
__NewHandle			Allocate a block in memory
Call with:	PushLong #	;Space for result	
	PushLong <i>BlockSize</i>	;Block size in bytes	
	PushWord <i>Owner</i>	;Program ID	
	PushWord <i>Attributes</i>	;Attributes of block	
	PushLong <i>Location</i>	;Location	
	__NewHandle		
Result:	Handle (longword)		
__DisposeHandle			Dispose of a block and its handle
Call with:	PushLong <i>Handle</i>	;Handle	
	__DisposeHandle		
Result:	None		
__DisposeAll			Discard all blocks and handles
Call with:	PushWord <i>Owner</i>	;Program ID	
	__DisposeAll		
Result:	None		
<hr/>			
Miscellaneous Tools			
__MTStartup			Start the Miscellaneous Tools
Call with:	__MTStartup		
Result:	None		
__MTShutDown			Shut down the Miscellaneous Tools
Call with:	__MTShutDown		
Result:	None		
__SysFailMgr			Exit and display an error message
Call with:	PushWord <i>DeathNum</i>	;Error code, follows message	
	PushLong <i>MessagePtr</i>	;Pointer to message string	
	__SysFailMgr		
Result:	None		
<hr/>			

Remember that before starting the Window Manager, you must do a LoadTools call to load the RAM-based tools your program uses. Remember also that you must allocate memory (with a NewHandle call) for the tool sets that need it.

Shut down the tool sets in the following order:

Desk Manager

Font Manager, Standard File, and other tool sets not listed here.

Menu Manager

Window Manager

Control Manager
 Dialog Manager
 Event Manager
 Line Editor
 QuickDraw II
 Miscellaneous Tools
 Memory Manager (after freeing all allocated memory)
 Tool Locator

Common Programming Errors

No matter how carefully you program, you may eventually run into a situation where the computer beeps and the program “crashes.” The beep indicates that the IIGS has executed a BRK instruction, which sends the processor into the Monitor. When that happens, all you can do is reboot. Needless to say, having a program crash gives you a helpless feeling — take it from one who knows.

What should you do when a program crashes? To begin, you should look for obvious errors in the program listing. Don’t bother looking for typographical errors; the assembler will have reported them when you assembled the program. Instead, look for errors related to tool calls. Among the most common are:

1. Calling a tool without activating it, *starting* its tool set, or misplacing the set’s StartUp call in the initialization sequence (see the preceding section).
2. Entering too few or too many input parameters, or entering them in the wrong order.
3. Pushing a word onto the stack when the tool expects a longword, or vice versa.
4. Pushing data onto the stack when the tool expects an address. Remember, *PushLong Loc* pushes the data stored at Loc, while *PushLong #Loc* pushes Loc’s address (i.e., it pushes a pointer to Loc).
5. Omitting a *PushWord #0* or *PushLong #0* call that reserves stack space for the result.

6. Forgetting to pull a result off the stack, or pulling 4 bytes when the tool returned only 2 (or vice versa).
7. Trying to exit the program without shutting down an active tool set. This error is easy to identify. The program will run fine, but crash upon exiting.
8. Failing to allocate enough working space in bank 0 with your New-Handle call. (See Table 6-1 for the memory requirements of the various tools sets.)

If the error isn't obvious from the program listing (or you don't want to take time to search the listing), trace through the program using the debugger, to see where the break occurred. The debugger displays instructions until the program makes a QDStartup call, at which point the program takes over and the screen displays your application. To make the instruction trace reappear, type `t` to give the debugger a "change to text mode" subcommand.

CHAPTER 7

Drawing with QuickDraw II

QuickDraw II is the tool set you use to display graphic images on the screen. Other tool sets also use QuickDraw II (or *QuickDraw*, for short) to draw, but they make calls to it behind the scenes, without your realizing it. This chapter introduces the Apple IIGS graphics facilities and guides you through the more important features of QuickDraw.

Graphics Modes

Earlier models of the Apple II provide four screen modes for displaying graphics. In each mode, the screen is divided into a grid of rows vertically and columns horizontally. The more rows and columns the grid has — that is, the higher its *resolution* — the sharper images will appear to the eye. The modes are:

- The *Low-Resolution* mode provides 48 rows and 40 columns, and lets you assign any of 16 colors to each coordinate (intersection of a row and column).
- The *High-Resolution* mode provides 192 rows and 280 columns, and

provides a “palette” of 8 colors. (Six colors, actually, because the palette includes 2 shades of both black and white.)

- The *double low-resolution* mode has (as you may have guessed) 92 rows and 80 columns, with 16 colors.
- The *double high-resolution* mode has 192 rows and 560 columns, also with 16 colors.

The Apple IIGS also supports these modes, although, like other Apple IIs, it has no ROM firmware for either double mode.

Unless you are creating a program to be run on other Apple IIs, you probably won't use any of these modes, because the Apple IIGS provides something better. Specifically, the IIGS provides two *super high-resolution* graphics modes. One mode divides the screen into 200 rows and 320 columns, and can display any of 16 colors at a given location. The other mode divides the screen into 200 rows and 640 columns, but lets you choose from only 4 colors for a given location.

I'll get back to these new modes later in this chapter, but first I must discuss the drawing environment in which they operate.

Drawing Environment

On earlier Apple IIs, your drawing space is limited to the coordinates in the screen grid. For example, the largest picture you can draw in regular high-resolution mode is one that occupies no more than 53,760 (280 by 192) screen locations. The Apple IIGS's super high-resolution graphics modes (or *super hi-res*, for short) can, of course, also display graphics that fill the screen, but with QuickDraw, you can *create* pictures that are much larger than the screen. To do this, you define the pictures in a so-called conceptual drawing space.

Conceptual Drawing Space

You use QuickDraw to create graphics images in a grid that is 32K (that is, 32,768) rows high and 32K columns wide. If you have worked with any other Apple II graphics mode, you might assume that QuickDraw's grid coordinates are numbered in the same way: with the row 0 and column 0 — or coordinate (0,0) — located at the top left-hand corner of the grid. That isn't the case here.

QuickDraw's 32K-by-32K grid consists of four 16K-by-16K quadrants,

with coordinate (0,0) at the center. Figure 7-1 shows how this grid, or *conceptual drawing space*, is numbered. Note that the row and column numbers are positive only in the lower right-hand quadrant; one or both numbers are negative in the other quadrants.

QuickDraw always assumes that you want to start working in the lower right-hand quadrant, so it “maps” onto the screen the image that extends down from and to the right of coordinate (0,0). (Of course, at the outset Quickdraw has no image to display. Your program must create it.) Thus, in the super hi-res 640 mode, the screen displays the graphics image that’s inside an imaginary rectangle whose top left-hand and bottom right-hand corners are at (0,0) and (199,639), respectively. On an Apple RGB color

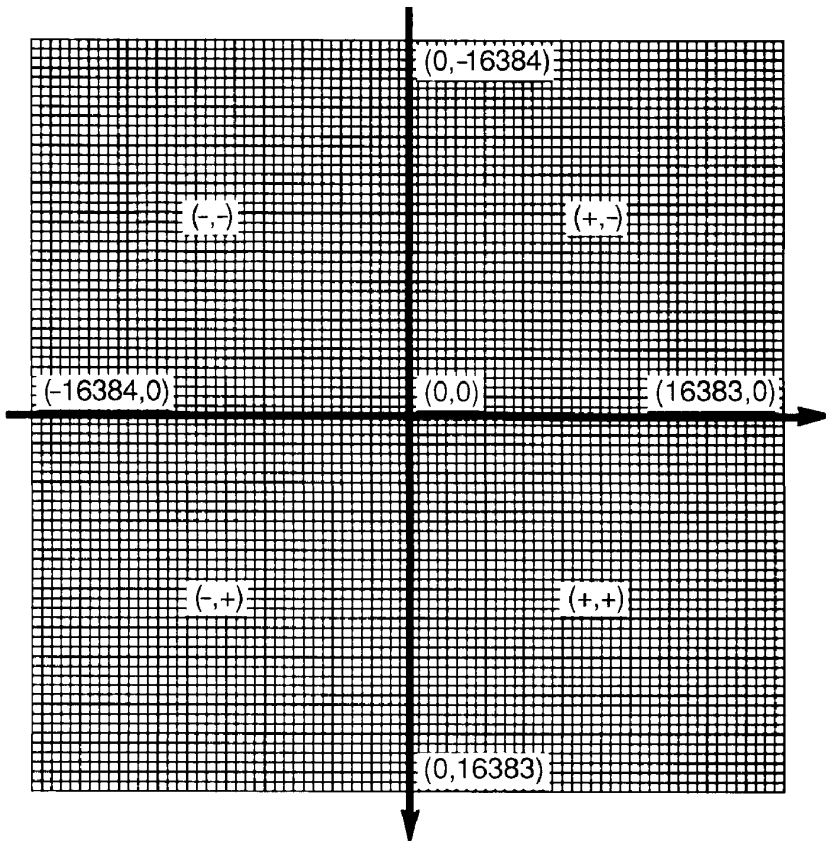


Figure 7-1

monitor, this image area — the entire screen minus the border — is about 6 inches high and 7 3/4 inches wide, so that's the "real" size of the imaginary rectangle.

By comparing these coordinates and dimensions to Figure 7-1, you can begin to realize how large the conceptual drawing space actually is, and what potential it offers for creating pictures. In 640 mode, for example, the screen can display only about five-hundredths of 1 percent of a quadrant, leaving space for almost 2,100 more screenfuls in that quadrant alone! In screen-size terms, this means that theoretically you could create a picture 25 feet high and 25 feet wide!

Be aware that the conceptual drawing space is neither in memory nor on disk. It is simply a coordinate system that is available for into which QuickDraw graphic data may be mapped. If your picture is too large to fit entirely in memory, your program should start by displaying the portion that fits on the screen, and read other portions from disk when the user scrolls to an area that's not visible. More about this later.

Pixels and Points

QuickDraw's drawing space is comprised of picture elements, or *pixels*, that appear as tiny dots on the screen. The screen displays 64,000 pixels in super hi-res 320 mode and 128,000 pixels in 640 mode.

The numeric coordinates of a specific pixel do not refer to the pixel itself, but to the *point* above and to the left of it. A point is simply the place where a row and column intersect — the "address" of a pixel within the conceptual drawing space. To appreciate the difference between a pixel and a point, see Figure 7-2.

Before you can draw something using a QuickDraw tool, you must tell QuickDraw at which point to put it. For example, to draw a rectangle, you must specify the row and column coordinates of its top left-hand and bottom right-hand corners. Here, the top corner indicates the rectangle's starting location and the bottom corner defines its size (see Figure 7-3).

QuickDraw Draws with a Pen

To actually see the rectangle, you must draw it with QuickDraw's *pen*. The pen is involved in every QuickDraw tool call that draws predefined shapes, such as rectangles, arcs, ovals, and lines. It's also involved in displaying text! QuickDraw's pen is more flexible than a regular office pen. Using it, you can draw lines of various thicknesses and even draw patterns.

If you have made the pen one pixel high and one pixel wide — that is,

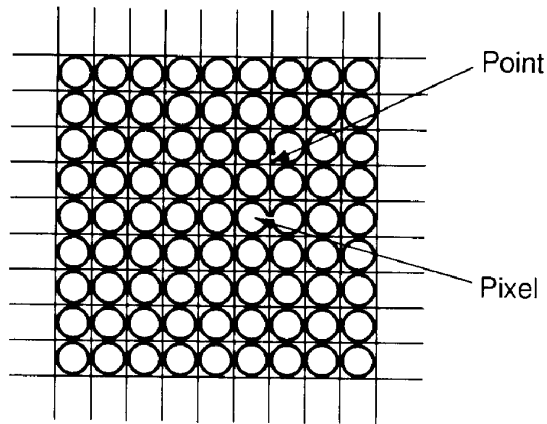


Figure 7-2

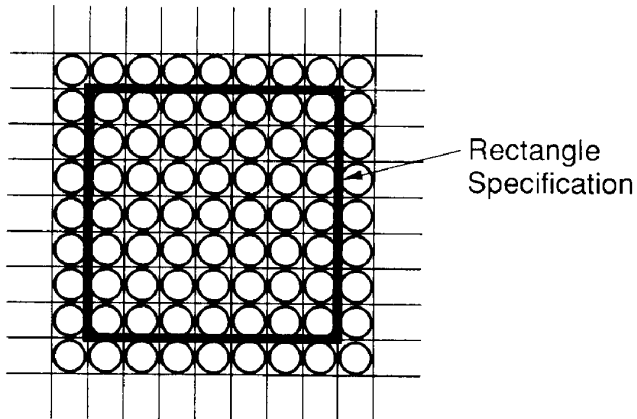


Figure 7-3

a single pixel — QuickDraw will draw the specified rectangle's one-pixel frame *inside* the imaginary rectangle defined by the corner coordinates. Figure 7-4 shows how such a rectangle would appear on the screen. (It would be much smaller, of course. To the eye, a pixel is only about the size of the period at the end of this sentence.)

Before you can use the pen, you must know how to start QuickDraw and shut it down. After all, you can't draw *anything* unless QuickDraw is active.

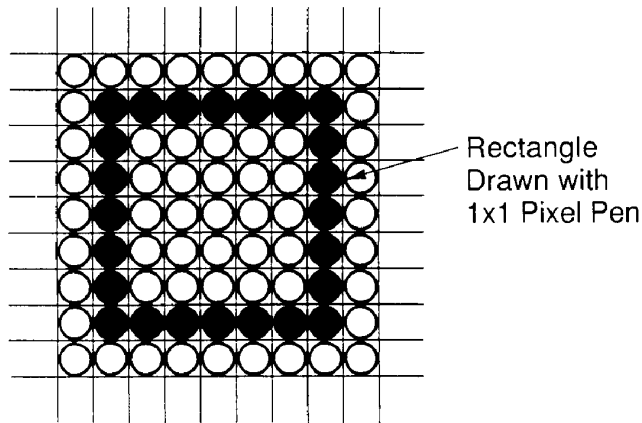


Figure 7-4

Starting and Stopping QuickDraw

To start QuickDraw, you must specify four things: the location of the direct page the Memory Manager has reserved for it, the super hi-res mode in which it is to operate (320 or 640), the maximum number of screen columns it can use for drawing, and the identification number of your program. The generalized sequence of the start-up call is:

```

PushWord DirectPageLoc    ;Direct page location
PushWord MasterSCB        ;Master scan line control byte
PushWord MaxWidth         ;Max. screen width (bytes)
PushWord ProgramID        ;The program's ID number
_QDStartup

```

If you use the program model from Chapter 6 (Example 6-2), you can obtain the direct page location for QuickDraw from the accumulator. Hence, the first call in the QDStartup calling sequence would be *pha*.

With the *master scan line control byte*, you tell QuickDraw which color table (more about color tables later) and which graphics mode to use (320 or 640) for drawing. Figure 7-5 shows the layout of the master scan line control byte. (The “fill” and “interrupt” bits relate to the computer’s video hardware; you can normally set them to zero.)

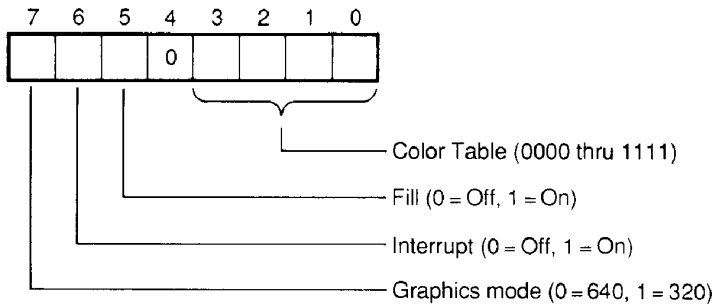


Figure 7-5

Generally, you want to use the default color table (0), so your second start-up call would be either *PushWord #0* (for 320 mode) or *PushWord #\$80* (for 640 mode). Better yet, by defining the scan line control byte with a global equate such as:

```
ScreenMode gequ $80 ;640 mode, no fill
```

as the model does, you can use *PushWord #ScreenMode*.

For most applications, you will want QuickDraw to be able to draw across the entire screen. To make it do this, use *PushWord #0* as your third call.

Finally, the program I.D. is the one returned by the Memory Manager when you start it. The model stores it in a variable named *MyID*, so your program's fourth start-up call would be *PushWord MyID*.

In summary, if you use the model, your QuickDraw start-up sequence is:

```
pha                ;Use DP from handle
PushWord #ScreenMode ;Graphics mode
PushWord #0         ;Use the entire screen
PushWord MyID       ;The program's ID number
__QDStartup
```

As with all tool sets, you must shut down QuickDraw when you finish using it. The call that does this is *__QDShutDown*.

The Pen

The pen with which QuickDraw draws has five properties, or *attributes*: location, size, mode, pattern, and mask. Collectively, they define the *pen state* (called *PenState* in tool calls).

Pen Location

The pen location (or *PenLoc*) specifies the point in the drawing space at which the pen is to sit ready to draw something. Understand, this is simply a location; the pen does not become visible until you actually start drawing with it.

Pen Size

The pen size (*PenSize*) defines the height and width of the pen's imaginary tip. That is, it determines how thick your lines will be. Unlike a regular pen, the tip on QuickDraw's pen is rectangular — so many pixels high by so many pixels wide. If you make the tip 4 pixels high and 2 pixels wide, for example, QuickDraw will draw horizontal lines twice as thick as vertical lines. You can also make the tip square, by specifying the same dimension for height and width. The most common size is 1 by 1 (to draw 1 pixel at a time), but you may want to make it, say, 4 by 4.

Pen size is measured relative to *PenLoc*. Height is measured down from it and width is measured to the right of it.

Pen Mode

QuickDraw provides eight different pen *modes*. The pen mode (*PenMode*) determines how a pen's pixels and its pixel pattern (described next) affect the existing pixels in an image when the pen draws over them.

For most applications, you can use the default mode, *Copy*, in which every pixel the pen draws overwrites (replaces) the pixel that's already at that location in the image. Apple has assigned a word-size number to each pen mode, and *Copy* is represented by \$0000. To learn about the other pen modes, refer to the *Apple IIGS Toolbox Reference*.

Pen Pattern

A regular pen, the kind people use in a home or office, always draws a solid line. You can use QuickDraw's pen in the same way, to draw solid lines or fill in shapes with a solid color. However, sometimes you may want to be

more creative. For example, you may want to draw a curly frame around a picture or give a textured look to the inside of a rectangle or circle.

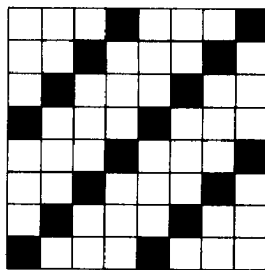
These kinds of artistic effects are easy to produce with QuickDraw; you simply specify the pattern you want the pen to reproduce. After that, it will apply that pattern to every line it draws and every shape it fills in. In short, assigning a pattern makes the pen repeat a specific design as it works its way across the screen, as if it were hanging wallpaper.

The format of the pen pattern (PenPat, in tool calls) differs between the two super hi-res modes. Let's consider the 320 mode first.

In 320 mode, the pen pattern is a data table that specifies the colors for an 8-by-8 square block of pixels. Each of the 64 entries in the table is a 4-bit number, or *nibble*, because in 320 mode, you can display 16 different colors. (Remember, a 4-bit number can have binary values from 0000 to 1111: 16 combinations in all.) Each number in the table indicates the color that the corresponding pixel is to have when the pen passes over it.

Assume for now we're drawing on a monochrome screen, with black images on a white background. In QuickDraw, black has the nibble value \$0 and white's value is \$F. Hence, in defining the table for a pen pattern, you would enter a 0 for pixels that the pen should turn "on" (display as black) and an F for pixels it should leave "off" (retain the white background).

Figure 7-6 shows a simple diagonal pattern you could create, and what it looks like when QuickDraw uses it to draw a rectangular frame. The PenPat table that defines this pattern is:



8x8 Diagonal
Pattern



Diagonal Pattern
As Pen Pattern

Figure 7-6

```

DiagPat  dc h'FFF0FFF0'    ;Row 0 (top)
          dc h'FF0FFF0F'    ;Row 1
          dc h'FOFFF0FF'    ;Row 2
          dc h'OFFF0FFF'    ;Row 3
          dc h'FFF0FFF0'    ;Row 4
          dc h'FF0FFF0F'    ;Row 5
          dc h'FOFFF0FF'    ;Row 6
          dc h'OFFF0FFF'    ;Row 7 (bottom)

```

For this particular pattern, you could use a small pen size and still obtain the diagonal effect. Even a 2-by-2 pen would do the job, although the diagonal might be rather difficult to see. With a 2-by-2 pen, Quickdraw will use the pattern's top two rows to draw horizontal lines and use its two left-hand columns to draw vertical lines.

You can't use a small pen size with all patterns, however. To fill a shape such as a rectangle or circle with a pattern, the pen must be thick enough to cover every significant pixel. Figure 7-7 shows a pattern that requires a thick pen (8-by-8, in this case), and what it looks like when used to fill a block on the screen. The PenPat table that produces this attractive cane-style motif is:

```

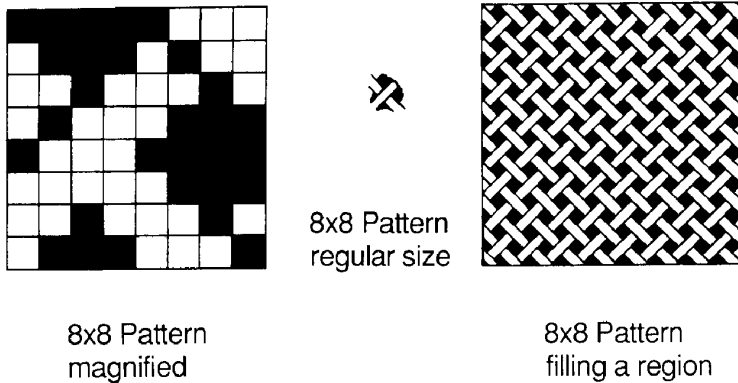
CanePat  dc h'FFFFFF00'    ;Row 0 (top)
          dc h'OFFF0F00'    ;Row 1
          dc h'00F000F0'    ;Row 2
          dc h'0F000FFF'    ;Row 3
          dc h'F000FFFF'    ;Row 4
          dc h'000F0FFF'    ;Row 5
          dc h'00F000F0'    ;Row 6
          dc h'OFFF000F'    ;Row 7 (bottom)

```

While QuickDraw's pen *always* draws using a pattern, you can make it draw with a solid color and, in effect, use no pattern at all. To do this, simply fill the PenPat table with the number of one specific color (e.g., \$0 for white).

In the super hi-res 640 mode, the PenPat table is also 256 bits long, but it defines colors for a block of 128 pixels (not 64, as in the 320 mode). This pixel block is 8 pixels high and 16 pixels wide.

Each entry in a 640 mode PenPat table is 2 bits long, because this mode can only display a pixel in one of four colors. (A discussion of colors is upcoming.) Here, black is numbered 0 and white is numbered 2.

**Figure 7-7**

Pen Mask

The pen mask (PenMask, in tool calls) lets you select which pixels in a pattern will be displayed when the pen draws. The mask is 64 bits long. In 320 mode, it has 1 bit for every pixel in the 8-by-8 pattern; in 640 mode, it has 1 bit for every *pair* of pixels in the 8-by-16 pattern. The rationale behind the 1-bit-per-pair layout relates to the way the 640 mode displays colors. More about that later.

Within the mask, each “1” bit permits the corresponding pixel (or pixel pair, in 640 mode) in the pattern to be displayed. Conversely, each “0” bit prevents its pattern counterpart from being displayed, and shows the background color instead. In short, the pen mask is a filter that selectively permits or blocks (1 or 0) the appearance of a pattern’s pixels.

Figure 7-8 shows a pen mask that blocks out every other pixel in a 320-mode pen pattern. In the pen mask shown here, the black boxes represent “1” (permit) bits and the white boxes represent “0” (block) bits.

You probably won’t want to apply a mask to very many patterns, and so you would fill it with 1’s, like this:

```
NoMask dc h'FF FF FF FF FF FF FF FF' ;No mask
```

Still, masks can be handy for using portions of a given pattern in several different drawing operations. For example, you may want to draw with the entire pattern at one place on the screen and with just the top half at another

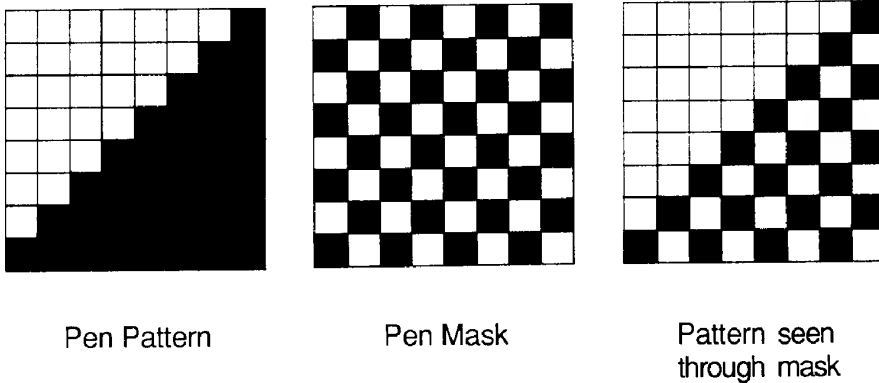


Figure 7-8

place. Preparing for this simply involves creating two masks (which is relatively easy), not two patterns (which is more difficult).

Pen State Tool Calls

Table 7-1 summarizes the tool calls you can use to read or set the pen state and its individual attributes. Note especially `PenNormal`, which sets the pen state (except for location) to QuickDraw's default values. `PenNormal` creates a 1-pixel pen that draws a solid white pattern with no mask. In the next section, I describe a few additional tool calls that govern the colors of pixels within the pen's pattern.

Colors

As mentioned earlier in this chapter, the two super hi-res graphics modes differ in the number of pixels and colors they display. One mode divides the screen into a pixel grid that is 200 rows high and 320 columns wide, and lets you display any pixel in 16 different colors. The other mode divides the screen into a grid of 200 rows and 640 columns, but lets you choose from only 4 pixel colors.

You're probably wondering why anyone would use a mode that offers only 4 colors. Actually, this mode is not as limiting as it sounds; you can get excellent color effects with it. Exactly *how* these effects are produced is not particularly intuitive, however, so I'll start by discussing the mode that

Table 7-1

Pen State Calls		
<code>__PenNormal</code>		Set the pen state to the normal values
Call with: <code>__PenNormal</code>		
Result: None		
Note: The standard pen state is <code>PenSize = 1,1</code> ; <code>PenMode = 0</code> (copy); <code>PenPat = black</code> ; <code>PenMask = 1</code> 's (no mask). The pen location is not changed.		
<code>__GetPenState</code>		Read the pen state
Call with: <code>PushLong PenState</code> ;Pointer to <code>PenState</code> buffer		
<code>__GetPenState</code>		
Result: None		
Note: <code>PenState</code> is:		
ds 4	;Row and column of <code>PenLoc</code>	
ds 4	;Width and height of <code>PenSize</code>	
ds 2	;Pen mode	
ds 32	;Pen pattern	
ds 8	;Pen mask	
<code>__SetPenState</code>		Set the pen state to the specified values
Call with: <code>PushLong PenState</code> ;Pointer to <code>PenState</code> record		
<code>__SetPenState</code>		
Result: None		
Note: <code>PenState</code> is:		
dc i'Vert,Horiz'	;Row and column of <code>PenLoc</code>	
dc i'Width,Height'	;Width and height of <code>PenSize</code>	
dc i'PenMode'	;Pen mode	
dc h'PenPattern'	;Pen pattern	
dc h'Pen mask'	;Pen mask	
Pen Location Calls		
<code>__GetPen</code>		Read the current pen location
Call with: <code>PushLong Point</code> ;Pointer to point variable		
<code>__GetPen</code>		
Result: None		
Note: The pen's coordinates are returned in <code>Point</code> , which should be defined as:		
Point ds 4		
GetPen will store the row number in the first 2 bytes and the column number in the last 2 bytes.		
<code>__MoveTo</code>		Move the pen to the specified point
Call with: <code>PushWord Horiz</code> ;Column		
<code>PushWord Vert</code> ;Row		
<code>__MoveTo</code>		
Result: None		
<code>__Move</code>		Move the pen by the specified displacements
Call with: <code>PushWord dHoriz</code> ;Columns to move		
<code>PushWord dVert</code> ;Rows to move		
<code>__Move</code>		
Result: None		

Table 7-1 (cont.)

Pen Size Calls		
<code>__GetPenSize</code>		Read the current pen size
Call with:	<code>PushLong <i>SizePtr</i></code> ;Pointer to <code>PenSize</code> loc.	
	<code>__GetPenSize</code>	
Result:	None	
Note:	<code>SizePtr</code> points to:	
	<code>ds 2</code> ;Width	
	<code>ds 2</code> ;Height	
<code>__SetPenSize</code>		Set the pen size
Call with:	<code>PushWord <i>Width</i></code>	
	<code>PushWord <i>Height</i></code>	
	<code>__SetPenSize</code>	
Result:	None	
Pen Pattern Calls		
<code>__GetPenPat</code>		Read the current pen pattern
Call with:	<code>PushLong <i>PatternBuf</i></code> ;Pointer to pattern buffer	
	<code>__GetPenPat</code>	
Result:	None	
Note:	<code>PatternBuf</code> is:	
	<code>ds 32</code>	
<code>__SetPenPat</code>		Set the pen pattern to the specified values
Call with:	<code>PushLong <i>PatternPtr</i></code> ;Pointer to pattern	
	<code>__SetPenPat</code>	
Result:	None	
Note:	<code>PatternPtr</code> points to:	
	<code>dc h'<i>Pattern</i>'</code>	
Pen Mask Calls		
<code>__GetPenMask</code>		Read the current pen mask
Call with:	<code>PushLong <i>MaskBuf</i></code> ;Pointer to mask buffer	
	<code>__GetPenMask</code>	
Result:	None	
Note:	<code>MaskBuf</code> is:	
	<code>ds 8</code>	
<code>__SetPenMask</code>		Set the pen mask to the specified values
Call with:	<code>PushLong <i>MaskPtr</i></code> ;Pointer to mask	
	<code>__SetPenMask</code>	
Result:	None	
Note:	<code>MaskPtr</code> points to:	
	<code>dc h'<i>Mask</i>'</code>	

displays 200-by-300 pixels in 16 colors — the so-called 320 mode — which is easier to understand.

320 Mode Colors

Recall that a 320 mode pen pattern consists of 4-bit values (nibbles) that select the color for each pixel in an 8-by-8 array. Each value is actually an index that QuickDraw uses to obtain color information from a 16-entry *color table*. The Apple IIGS provides 16 different color tables for 320 mode (and you can create others of your own), but most people use the default, or *standard*, color table.

Table 7-2 shows the standard color table for 320 mode (the one for 640 mode is different). Here, the Pixel Value column lists the hexadecimal digit that you would enter in your pen pattern and the Master Color Value column lists the 3-digit number that QuickDraw sends to the computer's color-generating circuitry.

The master color values may appear rather arbitrary, but by examining them closely you can appreciate how they relate to the colors they produce. For starters, note that black has the value 000, while white has the value FFF. This is appropriate, because black and white are, in fact, opposites. Black is the total absence of color, while white is a combination of the primary colors, red, green, and blue.

You can better understand the makeup of the master color values by examining the primary colors. Red has the value D00; its first digit is high-numbered digit (D) and the other digits are 0. Similarly, green (0E0) has a high-numbered second digit and 0 in the other positions, while blue (00F) has a high-numbered third digit and 0 in the other positions.

Table 7-2

Pixel Value	Color	Master Color Value
0 (\$0)	Black	000
1 (\$1)	Dark Gray	777
2 (\$2)	Brown	841
3 (\$3)	Purple	72C
4 (\$4)	Blue	00F
5 (\$5)	Dark Green	080
6 (\$6)	Orange	F70
7 (\$7)	Red	D00
8 (\$8)	Flesh	FA9
9 (\$9)	Yellow	FF0
10 (\$A)	Green	0E0
11 (\$B)	Light Blue	4DF
12 (\$C)	Lilac	DAF
13 (\$D)	Periwinkle Blue	78F
14 (\$E)	Light Gray	CCC
15 (\$F)	White	FFF

The conclusion is, then, *each digit in a master color value indicates the intensity of a primary color*. The first digit indicates how much red the pixel has, the middle digit how much green, and the third how much blue. For example, the value for yellow, FF0, tells you that it contains equal parts of red and green (with both at full intensity), but no blue.

640 Mode Colors

Recall that in 640 mode, the pixel values are only 2 bits long, allowing you to select from four colors (numbered 00, 01, 10, and 11). That being the case, you would expect the 640 mode's color table to be four entries long. Fortunately, it isn't; it has 16 entries, the same as in 320 mode.

You're probably wondering how a 2-bit number can be used to obtain color information from a 16-entry table. The fact is that the table consists of four *minipalettes*, each containing four master color values (see Table 7-3), from which a 2-bit pixel value selects a color. (Note that Table 7-3 shows the default color table for 640 mode. As with 320 mode, the Apple IIGS provides 15 additional tables. They are discussed at the end of this chapter.)

From which minipalette does QuickDraw obtain the color value for a pixel? When displaying graphics images, QuickDraw assigns 4 horizontally

Table 7-3

Pixel Value	Color	Master Color Value	
0	Black	000	<i>Minipalette 0</i>
1	Red	F00	
2	Green	0F0	
3	White	FFF	
0	Black	000	<i>Minipalette 1</i>
1	Blue	00F	
2	Yellow	FF0	
3	White	FFF	
0	Black	000	<i>Minipalette 2</i>
1	Red	F00	
2	Green	0F0	
3	White	FFF	
0	Black	000	<i>Minipalette 3</i>
1	Blue	00F	
2	Yellow	FF0	
3	White	FFF	

adjacent pixels to 4 consecutive minipalettes in the table. However, it does *not* access the minipalettes in the 0-1-2-3 order one might expect. Instead, *Quickdraw reads the color value of the 4 pixels from minipalettes 2, 3, 0, and 1, respectively.* It starts halfway down the table and “wraps around” to the beginning!

How can one produce worthwhile colored graphics using the 640 mode color table? After all, minipalettes 0 and 2 are identical, as are 1 and 3. Moreover, black and white appear in every minipalette, which doesn’t leave many entries open for additional colors. In short, QuickDraw provides only six colors: black, white, red, green, blue, and yellow. That’s not much to work with . . . or is it?

You *can*, in fact, produce dazzling color graphics in 640 mode by employing a display technique called *dithering*.

Dithering in 640 Mode

Early in the development of the IIGS, Apple’s engineers created a simple demonstration program that drew colored lines on the screen. It ran in 320 mode and generated a color for each line as a random 4-bit number between 0 and 15 (for the 16 entries in the color table). Rather than rewrite the entire program for the 640 mode, someone changed only the variable that selects the mode, then assembled the program and ran it.

Running in 640 mode, the program produced effects the engineers had not expected. Some lines showed one of the regular 640 mode colors, but others showed a shade produced when two of the regular colors were combined! In fact, there were *16* different shades in all.

Here’s what had happened. Aside from changing the mode select variable, the engineer who modified the program made no special provision for the fact that it was to run in 640 mode. That is, he or she did not allow for the difference in pen pattern formats between the modes. (Remember, 320 mode patterns consist of 4-bit values, but 640 patterns use 2-bit values.) As a result, while the 320 mode version of the program produced lines of one specific color from the table, the 640 mode version produced lines that combined two color values.

For example, if the program were to draw a line with color 6 (binary 0110) in 320 mode, it would use the seventh entry in the 320 mode color table — orange. However, receiving a 6 in 640 mode made the program break the 4-bit number into two parts and handle each part separately. Here, the program used the individual parts (01 and 10, respectively) to obtain color values from two different minipalettes. In doing this, it displayed the

first pixel as red and the second as yellow. Because pixels are extremely small on a 640 mode screen, when the red and yellow pixels were shown side by side, they appeared as the combination color, *orange*!

The important discovery that Apple's engineers made with their early 640-mode program was this: *when 2 pixels of different colors appear side by side, the eye sees them as a combination color, or shade*. This effect is called dithering.

By carefully selecting the pen pattern values your program draws with, you can produce the 16 shades that are available with dithering. Figure 7-9 summarizes these shades and the hex digit you must enter in a pen pattern to select each one. For example, to draw in blue, fill your pen pattern with the value 1; to draw in pastel green, fill it with B.

Tool Calls for Color

As you now know, the color or colors with which the pen draws depends on the values in the current pen pattern (PenPat). Table 7-1, shown earlier, lists three tool calls (PenNormal, SetPenState, and SetPenPat) that change the pen pattern, and two others (GetPenState and GetPenPat) that read the contents of the current pattern. Table 7-4 lists additional tool calls that involve the pen pattern, and thus, the color with which QuickDraw draws. This table also has calls that work with the screen's background pattern (or color).

When drawing with solid colors in 320-mode, you can use `SetSolidPenPat` and `SetSolidBackPat` to select your colors, and disregard the `SetPenPat` (described earlier) and `SetBackPat` calls. Here, the `ColorNum` input lets you specify any of the 16 colors (0 through 15) in the 320-mode color table.

SetSolidPenPat and **SetSolidBackPat** will also do the job in 640 mode, *provided* you only want one of the first four colors in the table — black, red, green, or white. To produce any other color, you must set up a dithered pattern (using the values in Figure 7-9) and call **SetPenPat** or **SetBackPat** to make QuickDraw use it. For example, to draw in blue on a 640-mode screen, you must first set the color with the following kind of sequence:

```

                                Pushlong #BluePenPat      ; Pens should
                                _SetPenPat                ; draw in blue
                                . .
                                . .
BluePenPat    dch'11 11 11 11 11 11 11 11 11 11'      ;Dithered
              dch'11 11 11 11 11 11 11 11 11 11'      ;blue
              dch'11 11 11 11 11 11 11 11 11 11'      ;pattern
              dch'11 11 11 11 11 11 11 11 11 11'
```

	Black	Blue	Yellow	White
Black	Black 0	Blue 1	Gold 2	Grey 3
Red	Red 4	Violet 5	Orange 6	Pink 7
Green	Green 8	Turquoise 9	Chartreuse A	Pastel Green B
White	Gray C	Pastel Blue D	Yellow E	White F

Figure 7-9

Drawing Lines, Rectangles, and Polygons

Now that you know how to set up the pen and work with its attributes, it's time to find out how to draw with it. QuickDraw has tool calls that let you draw lines, rectangles (with either square or rounded corners), polygons, ovals (or circles), and arcs. It also lets you draw *regions*: areas that contain several shapes. This section covers lines, rectangles, and polygons. Ovals, arcs, and regions are described in the next section.

QuickDraw always starts drawing at the current pen location and uses the current pen size, mode, pattern (with some exceptions), and mask. Hence, before drawing something, you must ensure that the pen has the attributes you want.

Table 7-4

Pen Pattern Calls	
__SetSolidPenPat	Set the pen pattern to a solid color
Call with: PushWord <i>ColorNum</i> ;Color number	
__SetSolidPenPat	
Result: None	
Note: QuickDraw only uses the appropriate number of bits in <i>ColorNum</i> . In 320 mode, it uses 4 bits, so <i>ColorNum</i> can range from 0 to 15. In 640 mode, it uses 2 bits, so <i>ColorNum</i> can be 0, 1, 2, or 3. In the default palette, this is black, red, green, and white, respectively.	
__SolidPattern	Set the specified pen pattern to a solid color
Call with: PushLong <i>PatternPtr</i> ;Pointer to pattern	
PushWord <i>ColorNum</i> ;Color number	
__SolidPattern	
Result: None	
Note: See note for SetSolidPenPat.	
Background Pattern Calls	
__GetBackPat	Read the background pattern
Call with: PushLong <i>PatternPtr</i> ;Pointer to pattern buffer	
__GetBackPat	
Result: None	
Note: <i>PatternPtr</i> points to: ds 32	
__SetBackPat	Set the background to the specified pattern
Call with: PushLong <i>PatternPtr</i> ;Pointer to pattern	
__SetBackPat	
Result: None	
Note: <i>PatternPtr</i> points to: dc h' <i>Pattern</i> '	
__ClearScreen	Clear the screen to the specified color
Call with: PushWord <i>ColorWord</i>	
__ClearScreen	
Result: None	
Note: To obtain a solid color on the screen, all 4 hex digits in <i>ColorWord</i> must be identical. For example, to set a 320 mode screen to light blue, enter PushWord #5BBB5B .	

Lines

A line is the easiest object to draw. As Table 7-5 shows, there are only two line-drawing calls: **LineTo** and **Line**. They're similar, but with **LineTo** you specify the end point of the line, whereas with **Line** you specify displacements relative to the starting point.

Table 7-5

<hr/>		
<code>_LineTo</code>		Draw a line to the specified point
Call with:	<code>PushWord Horiz ;Column</code> <code>PushWord Vert ;Row</code> <code>_LineTo</code>	
Result:	None	
<hr/>		
<code>_Line</code>		Draw a line to the point specified by displacements
Call with:	<code>PushWord dHoriz ;End is dHoriz columns</code> <code>PushWord dVert ; and dVert rows away</code> <code>_Line</code>	
Result:	None	
<hr/>		

To draw a line, move the pen to its starting point and give a `LineTo` call. For example, to connect points (10,20) and (30,40), enter:

```

PushWord #20      ;Specify the starting column
PushWord #10      ; and row positions
_MoveTo          ;Move the pen there
PushWord #40      ;Specify the ending column
PushWord #30      ; and row positions
_LineTo

```

Note that a `Line` call with 20 for both the horizontal and vertical displacement would do the same thing as the preceding `LineTo` call. `Line`'s parameters are signed values, so you can specify negative as well as positive displacements. For example, a vertical displacement of `-10` sets the end point 10 rows above the starting point.

Rectangles

QuickDraw has four tool calls for drawing rectangles (see table 7-6). For each, you must supply a pointer to the rectangle's top left-hand corner and bottom right-hand corner coordinates. This is a data structure of the form:

```

dc i' V1, H1'      ;Row, column of top left-hand corner
dc i' V2, H2'      ;Row, column of bottom right-hand corner

```

The first call, `FrameRect`, draws the boundary or *frame* of a rectangle and shows the inside in the background pattern. For example, the following

Table 7-6

<code>__FrameRect</code>	Draw the boundary of the specified rectangle
Call with: <code>PushLong RectPtr</code>	;Pointer to rectangle definition
	<code>__FrameRect</code>
Result:	None
Note:	<code>RectPtr</code> points to:
	<code>dc i'V1,H1'</code> ;Row, column of top left-hand corner
	<code>dc i'V2,H2'</code> ;Row, column of bottom right-hand corner
<code>__PaintRect</code>	Fill the interior of a rectangle with the current pen pattern
Call with: <code>PushLong RectPtr</code>	;Pointer to rectangle definition
	<code>__PaintRect</code>
Result:	None
Note:	See <code>__FrameRect</code> for the rectangle definition.
<code>__FillRect</code>	Fill the interior of a rectangle with the specified pen pattern
Call with: <code>PushLong RectPtr</code>	;Pointer to rectangle definition
	<code>PushLong PatternPtr</code> ;Pointer to pattern
Call with: <code>__FillRect</code>	
Result:	None
Note:	See <code>__FrameRect</code> for the rectangle definition. <code>PatternPtr</code> points to:
	<code>dc h'Pattern'</code>
<code>__EraseRect</code>	Fill the interior of a rectangle with the background color
Call with: <code>PushLong RectPtr</code>	;Pointer to rectangle definition
	<code>__EraseRect</code>
Result:	None
Note:	See <code>__FrameRect</code> for the rectangle definition.

draws the frame of a rectangle whose top left-hand corner is at (10,20) and bottom right-hand corner is at (40,60):

```

                PushLong #Rect1    ;Pointer to rectangle
                __FrameRect        ; coordinates
                . .
                . .
Rect1  dc i'10,20'                ;Top left-hand corner
       dc i'40,60'                ;Bottom right-hand corner

```

You can use two other calls to draw solid rectangles. `PaintRect` draws a solid rectangle using the current pen pattern, while `FillRect` draws one using a pen pattern that you specify. Finally, the `EraseRect` call removes a rectangle from the screen by replacing it with the background pattern.

Be aware, incidentally, that any solid rectangle (or other shape) you draw replaces whatever is previously displayed within its boundary. Thus, to draw one solid rectangle inside another, always draw the outer rectangle first. If you draw the inner one first, it won't appear on the screen; the outer rectangle will obliterate it.

By now, you're probably tired of reading about tool calls and want to see some of them used in an actual program. That's next on the agenda.

A Program that Draws Rectangles

This section presents a program that draws four rectangles. Before looking at the actual listing, let's see what the program does and what it produces on the screen.

When I first started to think about this example program, I decided that it should include a mixture of tool calls that illustrate not only the use of `FrameRect` and `PaintRect`, but show the use of some pen-related tool calls as well. To start working on the program, I drew the rectangles on a piece of graph paper and named them `Rect1`, 2, 3, and 4.

Figure 7-10 is similar to (but much neater than) my original drawing. Note that `Rect1` and `Rect3` are frames or boundaries, while `Rect2` and `Rect4` are solid. The black borders that enclose `Rect1` represent the default background color, black.

Here, then, are the specifications for my four rectangles:

- *Rect1* is a frame whose top left and bottom right corners are at (70,90) and (155,310). It is drawn with the default pen size (1,1) and a solid pen pattern of color 5. This dither of red (binary 01) and blue (01) produces violet.
- *Rect2* is a solid rectangle whose top left and bottom right corners are at (75,100) and (150,300). It is drawn with the default pen size (1,1) and a solid pattern of color \$F, white.
- *Rect3* is a frame whose top left and bottom right corners are at (60,75) and (165,320). It is drawn with a pen size of (5,5) and a solid pattern of color 2. This produces a dither of black (00) and green (10), which appears as green.
- *Rect4* is a solid rectangle whose top left and bottom right corners are at (60,375) and (165,620). It is drawn with a pen size of (5,5) and a solid pen pattern of color 5. As with `Rect1`, this dither of red (01) and blue (01) produces violet.

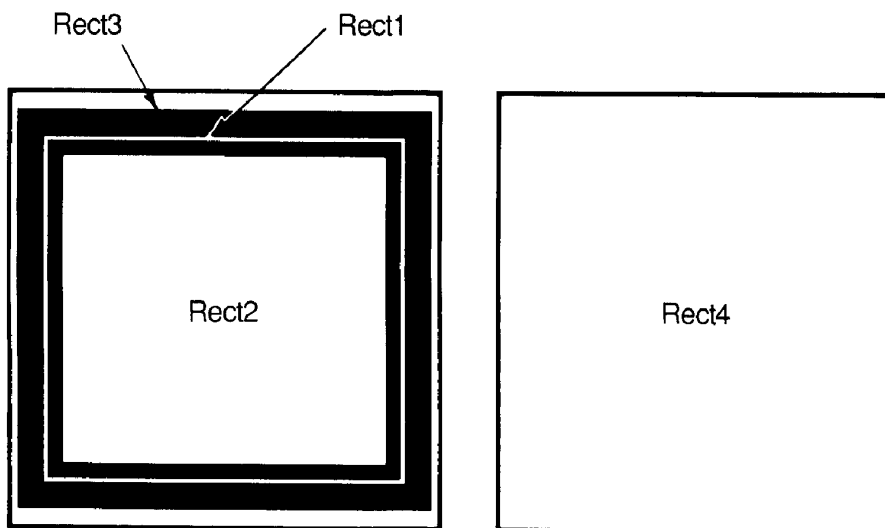


Figure 7-10

My final consideration was the mode. Drawing in 320 mode would have made my job somewhat easier, but I chose 640 mode instead, to illustrate the dithering effect.

Example 7-1 lists my final rectangle-drawing program, RECTS, which has two data segments and four code segments. One data segment, Global-Data, contains variables used throughout the program; the other, RectData, contains the rectangle coordinates and a pen state record.

The first code segment, Rects, simply keeps a global equate separate from other code. The DoIt segment is the program's "main lobby." It calls the InitStuff subroutine to start up the tools, DrawRects to draw the rectangles, and EventLoop to wait for the user's signal to quit. Finally, tool calls at the end of DoIt shut down the tools and exit.

Note that the user interface segment, EventLoop, contains three instructions that keep the rectangles on the screen until the user presses a key. This is an Apple II way of waiting for a key. You can accomplish the same thing with tool calls, but since I have not yet introduced those calls, I can't, in fairness, use them.

Polygons

Polygons are shapes that have three or more sides, such as triangles, hexagons, and octagons. To display a polygon, you must define its shape, then

Example 7-1

; RECTS draws four rectangles in 640 mode.

```

        absaddr on
        MCOPY Rects.macros

Rects      START
           using GlobalData

;-----
;
; Global equate used throughout the program.

ScreenMode    gequ $80                ;640 mode, no fill

           phk                        ;Set data bank to program
           plb                        ; bank to allow absolute addressing

           jsr InitStuff              ;Initialize everything
           bcs AllDone               ;Quit if initialization fails

           jsr DrawRects              ;Draw the rectangles

           jsr EventLoop              ;Wait for user to press a key

AllDone       anop                    ;All is done, shut down
               _QDShutDown            ;QuickDraw II
               _MTShutDown            ;Miscellaneous Tools

           PushWord MyID              ;Discard the program's handle
               _DisposeAll

           PushWord MyID              ;Memory manager
               _MMShutdown            ;Tool Locator
               _TLShutdown

               _Quit QuitParams       ;Do a ProDOS Quit call
               brk $F0                ;If it fails, break

           END

;*****
;
; Global Data
;
;*****

GlobalData    DATA

QuitParams    dc i4'0'                ;Return to caller
               dc i'$4000'            ;Make program restartable in memory

MyID          ds 2                    ;This will hold the program's i.d.

           END

```

216 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

```

;*****
;
; InitStuff
;
; Initializes tool sets and gets space in bank 0 for use as
; direct page by tools that need it.
;
;*****

InitStuff      START
               using GlobalData

; Initialize the Tool Locator, Memory Manager, and Miscellaneous
; Tools.

               _TLStartup

               PushWord #0
               _MMStartup

               pla                               ;Memory Manager returns program's ID
               sta MyID

               _MTStartup
               ldx #3
               jsr PrepareToDie

; Get some space for the zero page we need. QuickDraw needs
; three pages.

               PushLong #0                      ;Space for handle
               PushLong #$300                   ;Three pages for QuickDraw
               PushWord MyID                    ;Owner
               PushWord #$C005                  ;Locked, fixed, fixed bank
               PushLong #0                      ;Location
               _NewHandle
               ldx #$FF
               jsr PrepareToDie

               pla                               ;Read handle
               sta 0                            ; and store in direct page
               pla
               sta 2

               lda [0]                          ;Get DP location from handle

; Initialize QuickDraw

               pha                               ;Use dp obtained from handle
               PushWord #ScreenMode             ;Mode = 640
               PushWord #0                      ;Use entire screen
               PushWord MyID
               _QDStartup
               ldx #4
               jsr PrepareToDie

               clc                               ;Clear the carry flag
               rts                              ; and return

END

```

```

;*****
;
; Draw the rectangles
;
;*****

DrawRects START
    using RectData

    PushWord #5                ;Set the pen color
    _SetSolidPenPat

    PushLong #Rect1            ; and draw the Rect1 frame
    _FrameRect

    PushWord #$F               ;Now switch to white
    _SetSolidPenPat

    PushLong #Rect2            ; and paint Rect2
    _PaintRect

; Change pen attributes one at a time

    PushWord #5                ;Size is 5,5
    PushWord #5
    _SetPenSize

    PushWord #0                ;Regular mode
    _SetPenMode

    PushWord #2                ;Pattern
    _SetSolidPenPat

    PushLong #PenMask           ;Mask
    _SetPenMask

; Let's see what the frame looks like

    PushLong #Rect3            ;Draw Rect3 frame
    _FrameRect

; Now an example of how to set up everything at once

    PushLong #PenRec
    _SetPenState

    PushLong #Rect4            ;Paint Rect4
    _PaintRect

    rts
    END

;*****
;
; Event Loop
;
; Keep the rectangles on the screen until user presses a key.
;
;*****

```

218 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

EventLoop START

```
WaitHere        lda $E0C000
                 bpl WaitHere
                 lda $E0C010
                 rts
```

END

```
;*****
;
; Rectangle Data
;
; Coordinates of the rectangles and a pen state record.
;
;*****
```

RectData DATA

```
Rect1    dc i'70'                                ;Top left corner is at
           dc i'90'                                ; 70,90
           dc i'155'                              ;Bottom right corner is at
           dc i'310'                              ; 155,310

Rect2    dc i'75'                                ;Top left corner is at
           dc i'100'                              ; 75,100
           dc i'150'                              ;Bottom right corner is at
           dc i'300'                              ; 150,300

Rect3    dc i'60'                                ;Top left corner is at
           dc i'75'                                ; 60,75
           dc i'165'                              ;Bottom right corner is at
           dc i'320'                              ; 165,320

Rect4    dc i'60'                                ;Top left corner is at
           dc i'375'                              ; 60,375
           dc i'165'                              ;Bottom right corner is at
           dc i'620'                              ; 165,620
```

```
PenRec    anop
PenLoc    dc i'0,0'                              ;Pen location
PenSize   dc i'5,5'                              ;Pen size
PenMode   dc i'0'                                ;Pen mode
PenPat    dc h'55 55 55 55 55 55 55 55'        ;Pen Pattern
           dc h'55 55 55 55 55 55 55 55'
           dc h'55 55 55 55 55 55 55 55'
           dc h'55 55 55 55 55 55 55 55'
PenMask   dc h'FF FF FF FF FF FF FF FF'        ;No mask
```

END

```
*****
*
* Prepare To Die
*
* Dies if carry is set.
*
* Error number to display is in X register.
* Assumes that Miscellaneous Tools is active.
*
*****
```

```

PrepareToDie  START
               bcs RealDeath      ; Carry = 1?
               rts                 ; No. Return to caller

RealDeath     phx                  ; Yes. Goodbye, program.
               PushLong #DeathMsg
               _SysFailMgr

DeathMsg      str 'Could not handle error '

               END

```

draw it. Defining the shape involves entering a series of MoveTo and LineTo calls that specify how the pen should move. To tell QuickDraw how far the definition extends, you must enter an OpenPoly call at the beginning of it and a ClosePoly call at the end. Table 7-7 summarizes OpenPoly and ClosePoly and the calls that actually draw or erase a polygon.

Note that just as the Memory Manager assigns a handle to your program when you call NewHandle, QuickDraw assigns a handle to a polygon definition when you call OpenPoly. This handle is what you need to draw or erase the polygon. You must also discard the handle (using DisposHandle) at the end of your program, just as you must discard the program's handle.

Example 7-2 lists a program called POLY that paints a right triangle (one that contains a 90-degree angle) in 640 mode at the bottom of the screen. Like the earlier RECTS program, POLY includes Apple II-style instructions that keep the figure on the screen until the user presses a key.

Drawing Other Shapes

QuickDraw can also draw ovals, arcs, regions, and rectangles with round corners (*roundrects*). Most of the tool calls for these shapes are RAM-based, as opposed to the calls that draw lines, rectangles, and polygons, which are in ROM. To run a program that calls a RAM-based tool, you must start the computer by booting up the System Disk (that loads the RAM-based tools in memory).

Recall from Chapter 6 that there is a set of "MountBookDisk calls that prompt the user to insert the System Disk if he or she has violated this rule. The program model includes these calls and so should your programs.

This section describes only how to draw ovals (including circles) and regions. Arcs and roundrects are generally less useful for application programs, but if you need them, refer to Apple's documentation.

Table 7-7

Definition Calls		
<hr/>		
__OpenPoly		Get a handle for a polygon data structure
Call with:	PushLong # __OpenPoly	;Space for result (handle)
Result:	Handle (long word)	
Note:	OpenPoly begins the definition of a polygon. (Define the polygon with MoveTo and LineTo calls.) ClosePoly ends it.	
__ClosePoly		End the current polygon definition
Call with:	__ClosePoly	
Result:	None	
<hr/>		
Drawing Calls		
<hr/>		
__FramePoly		Draw the boundary of the specified polygon
Call with:	PushLong <i>PolyHandle</i> __FramePoly	;Polygon handle
Result:	None	
__PaintPoly		Fill the interior of a polygon with the current pen pattern
Call with:	PushLong <i>PolyHandle</i> __PaintPoly	;Polygon handle
Result:	None	
__FillPoly		Fill the interior of a polygon with the specified pen pattern
Call with:	PushLong <i>PolyHandle</i> PushLong <i>PatternPtr</i>	;Polygon handle ;Pointer to pattern
Result:	None	
__ErasePoly		Erase the specified polygon
Call with:	PushLong <i>PolyHandle</i> __ErasePoly	;Polygon handle
Result:	None	
<hr/>		

Ovals

Anyone who has taken geometry class might expect an oval-drawing operation to involve arcs and angles and the like. Fortunately, that's not the case with QuickDraw. As Table 7-8 shows, to draw an oval, you simply specify the corner points of the *rectangle* in which the oval is to be inscribed.

For example, to produce an oval that is 30 pixels high and 15 pixels wide, you would specify a rectangle of that size when you make your oval-drawing call. You can also use the oval calls to draw *circles*, by making your rectangle square.

Example 7-2

; POLY draws a triangle in 640 mode.

```

        absaddr on
        MCOPY Poly.macros

Poly          START
              using GlobalData

;-----
;
; Global equate used throughout the program.

ScreenMode    gequ $80                ;640 mode, no fill

              phk                      ;Set data bank to program
              plb                      ; bank to allow absolute addressing

              jsr InitStuff            ;Initialize everything
              bcs AllDone             ;Quit if initialization fails

              jsr DrawPoly            ;Draw the triangle

              jsr EventLoop           ;Wait for user to press a key

AllDone        anop                    ;All is done, shut down
              _QDShutDown             ;QuickDraw II
              _MTShutDown             ;Miscellaneous Tools

              PushWord MyID           ;Discard the program's handles
              _DisposeAll

              PushWord MyID           ;Memory Manager
              _MMShutDown             ;Tool Locator
              _TLShutDown

              _Quit QuitParams        ;Do a ProDOS Quit call
              brk $F0                 ;If it fails, break

              END

;*****
;
; Global Data
;
;*****

GlobalData    DATA

TriangleH     ds 4                    ;This will hold triangle's handle

QuitParams    dc i4'0'                ;Return to caller
              dc i'$4000'             ;Make program restartable in memory

MyID          ds 2                    ;This will hold the program's i.d.

              END

```

222 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

```
;*****
;
; InitStuff
;
; Initializes tool sets and gets space in bank 0 for use as
; direct page by tools that need it.
;
;*****

InitStuff      START
               using GlobalData

; Initialize the Tool Locator, Memory Manager, and Miscellaneous
; Tools.

        _TLStartup

        PushWord #0
        _MMStartup

        pla                                ;Memory Manager returns program's ID
        sta MyID

        _MTStartup
        ldx #3
        jsr PrepareToDie

; Get some space for the direct page we need. QuickDraw needs
; three pages.

        PushLong #0                      ;Space for handle
        PushLong #$300                   ;Three pages for QuickDraw
        PushWord MyID                    ;Owner
        PushWord #$C005                  ;Locked, fixed, fixed bank
        PushLong #0                      ;Location
        _NewHandle
        ldx #$FFF
        jsr PrepareToDie

        pla                                ;Read handle
        sta 0                            ; and store in direct page
        pla
        sta 2

        lda [0]                          ;Get dp location from handle

; Initialize QuickDraw

        pha                                ;Use dp obtained from handle
        PushWord #ScreenMode             ;Mode = 640
        PushWord #0                      ;Use entire screen
        PushWord MyID
        _QDStartup
        ldx #4
        jsr PrepareToDie

        clc                                ;Clear the carry flag
        rts                               ; and return

END
```



```

;*****
;
; Define the polygon, then draw it
;
;*****

DrawPoly  START
          using GlobalData

; This is the polygon's definition

          PushLong #0                ;Space for result (handle)
          _OpenPoly                  ;Start the definition
          pla                        ;Read the handle into TriangleH
          sta TriangleH
          pla
          sta TriangleH+2

          PushWord #100              ;Move pen to starting point
          PushWord #100              ; (100,100)
          _MoveTo

          PushWord #100              ;Draw a vertical line
          PushWord #200              ; from (100,100) to (200,100)
          _LineTo

          PushWord #250              ;Draw a horizontal line
          PushWord #200              ; from (200,100) to (200,250)
          _LineTo

          PushWord #100              ;Draw a line back to the
          PushWord #100              ; starting point
          _LineTo

          _ClosePoly                ;End the polygon definition

; Set up the pen and draw the triangle

          PushWord #5                ;Set the pen color
          _SetSolidPenPat

          PushLong TriangleH         ; and paint the triangle
          _PaintPoly

          rts
          END

;*****
;
; Event Loop
;
; Keep the triangle on the screen until user presses a key.
;
;*****

EventLoop  START

WaitHere   lda $E0C000
           bpl WaitHere

```

```

        lda $E0C010
        rts

END

*****
*
* Prepare To Die
*
* Dies if carry is set.
*
* Error number to display is in X register.
* Assumes that Miscellaneous Tools is active.
*
*****

PrepareToDie  START
               bcs RealDeath      ;Carry = 1?
               rts                ; No. Return to caller

RealDeath     phx                  ; Yes. Goodbye, program.
               PushLong #DeathMsg
               __SysFailMgr

DeathMsg      str 'Could not handle error '

END

```

Table 7-8

<u>__FrameOval</u>	Draw the boundary of an oval inscribed in the specified rectangle
Call with: PushLong <i>RectPtr</i>	;Pointer to rectangle definition
	<u>__FrameOval</u>
Result:	None
Note:	RectPtr points to:
	dc i' V1,H1' ;Row, column of top left-hand corner
	dc i' V1,H2' ;Row, column of bottom right-hand corner
<u>__PaintOval</u>	Fill the interior of an oval with the current pen pattern
Call with: PushLong <i>RectPtr</i>	;Pointer to rectangle definition
	<u>__PaintOval</u>
Result:	None
Note:	See <u>__FrameOval</u> for the rectangle definition.
<u>__FillOval</u>	Fill the interior of an oval with the specified pen pattern
Call with: PushLong <i>RectPtr</i>	;Pointer to rectangle definition
	PushLong <i>PatternPtr</i> ;Pointer to pattern
	<u>__FillOval</u>
Result:	None
Note:	See <u>__FrameOval</u> for the rectangle definition. PatternPtr points to:
	dc h' <i>Pattern</i> '

Table 7-8 (cont.)

<code>__EraseOval</code>	Fill the interior of an oval with the background color
Call with: <code>PushLong RectPtr</code>	;Pointer to rectangle definition
<code>__EraseOval</code>	
Result: None	
Note: See <code>__FrameOval</code>	for the rectangle definition.

Regions

A region is a rectangular area that encloses one or more shapes. Once you have defined the shapes in the region, you can make QuickDraw display all of them at once by telling it to draw the region.

Defining a region is similar to defining a polygon; that is, you begin with an `OpenRgn` call, enter the drawing calls that define the shapes in the region, and end the definition with a `CloseRgn` call. However, because a region can hold several shapes, you must allocate space in memory for it with a `NewRgn` call. (It is `NewRgn`, not `OpenRgn`, that returns the region's handle.) The `CloseRgn` call at the end of the definition creates the region and frees the memory it uses. In summary, then, here is the procedure to create a region:

1. Call `NewRgn` to allocate memory for the region. `NewRgn` returns the region's handle.
2. Call `OpenRgn` to start defining the region.
3. Draw the shapes in the region. The legal calls here are `MoveTo`, `Line`, `LineTo`, `FrameRect`, `FrameOval`, `FrameRRect` (frame a roundrect), `FramePoly`, and `FrameRgn`.
4. Call `CloseRgn` to end the definition and free the allocated memory.
5. Draw the region with a `FrameRgn` or `PaintRgn` call.

Table 7-9 summarizes the tool calls most commonly used for regions.

Suppose, for example, you want to display a red clown-style bow tie (two triangles with a circular "knot" between them) that is 30 pixels high and 100 pixels wide. If you want the top left-hand point of the tie to appear at (30,30), Figure 7-11 shows the coordinates your program needs to draw it.

To produce the bow tie, the program must set up a region for it, make

Table 7-9

Definition Calls		
<hr/>		
__NewRgn		Allocate space for a new structure
Call with:	PushLong #0	;Space for result (handle)
	__NewRgn	
Result:	Handle (longword)	
Note:	CloseRgn frees the allocated space.	
__OpenRgn		Start defining a region
Call with:	OpenRgn	
Result:	None	
Note:	CloseRgn ends the region definition.	
__CloseRgn		End the current region definition
Call with:	PushLong <i>RgnHandle</i>	;Region handle
	__CloseRgn	
Result:	None	
<hr/>		
Drawing Calls		
__FrameRgn		Draw the boundary of the specified region
Call with:	PushLong <i>RgnHandle</i>	;Region handle
	__FrameRgn	
Result:	None	
__PaintRgn		Fill the interior of a region with the current pen pattern
Call with:	PushLong <i>RgnHandle</i>	;Region handle
	__PaintRgn	
Result:	None	
__FillRgn		Fill the interior of a region with the specified pen pattern
Call with:	PushLong <i>RgnHandle</i>	;Region handle
	PushLong <i>PatternPtr</i>	;Pointer to pattern
	__FillRgn	
Result:	None	
__EraseRgn		Erase the specified region
Call with:	PushLong <i>RgnHandle</i>	;Region handle
	__EraseRgn	
Result:	None	

`LineTo` calls to define the triangular bows and a `FrameOval` call to define the knot, and give a region-drawing call make `QuickDraw` draw the region. Example 7-3 lists the program, called BOWTIE. Note that I had to load `QuickDraw`'s RAM-based tools from disk (with `LoadOneTool`), to obtain the `FrameOval` call. If the region had no oval, that wouldn't have been necessary.

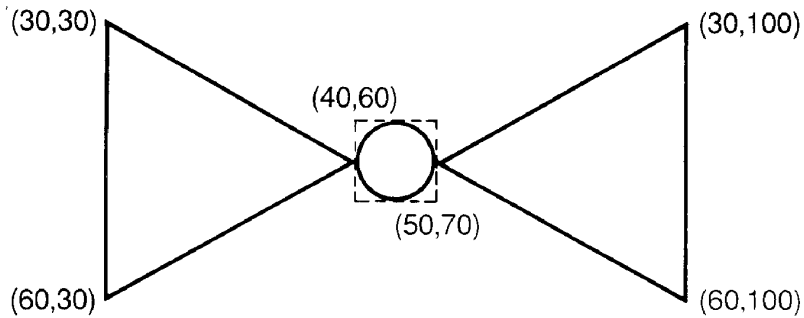


Figure 7-11

Example 7-3

; BOWTIE draws a region that defines a bowtie shape.

```
absaddr on
MCOPY Bowtie.macros
```

```
Bowtie      START
            using GlobalData
```

```
;-----
;
```

; Global equate used throughout the program.

```
ScreenMode    gequ $80                ;640 mode, no fill

phk            ;Set data bank to program
plb            ; bank to allow absolute addressing

jsr InitStuff    ;Initialize everything
bcs AllDone      ;Quit if initialization fails

jsr DrawTie      ;Draw the bowtie

jsr EventLoop    ;Wait for user to press a key

AllDone        anop                    ;All is done, shut down
                _QDShutDown            ;QuickDraw II
                _MTShutDown            ;Miscellaneous Tools

                PushWord MyID           ;Discard the program's handles
                _DisposeAll

                PushWord MyID           ;Memory Manager
                _MMShutDown            ;Tool Locator
                _TLShutDown

                _Quit QuitParams        ;Do a ProDOS Quit call
                brk $F0                ;If it fails, break

END
```

228 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

```
;*****
;
; Global Data
;
;*****

GlobalData    DATA

TieHandle     ds 4                ;This will hold bowtie's handle

KnotDef       dc i'40,60'        ;Rectangle that encloses knot
              dc i'50,70'

QuitParams    dc i4'0'           ;Return to caller
              dc i'$4000'        ;Make program restartable in memory

MyID          ds 2                ;This will hold the program's i.d.

VolNotFound   equ $45            ;ProDOS error

END

;*****
;
; InitStuff
;
; Initializes tool sets and gets space in bank 0 for use as
; direct page by tools that need it.
;
;*****

InitStuff     START
              using GlobalData

; Initialize the Tool Locator, Memory Manager, and Miscellaneous
; Tools.

        _TLStartup

        PushWord #0
        _MMStartup

        pla                                ;Memory Manager returns program's ID
        sta MyID

        _MTStartup
        ldx #3
        jsr PrepareToDie

; Get some space for the direct page we need. QuickDraw needs
; three pages.

        PushLong #0                      ;Space for handle
        PushLong #$300                  ;Three pages for QuickDraw
        PushWord MyID                   ;Owner
        PushWord #$C005                 ;Locked, fixed, fixed bank
        PushLong #0                     ;Location
        _NewHandle
        ldx #$FF
        jsr PrepareToDie
```

```

        pla                                ;Read handle and store in DP
        sta 0
        pla
        sta 2

        lda [0]                            ;Get dp location from handle

; Initialize QuickDraw

        pha                                ;Use dp obtained from handle
        PushWord #ScreenMode               ;Mode = 640
        PushWord #0                        ;Use entire screen
        PushWord MyID
        _QDStartup
        ldx #4
        jsr PrepareToDie

;-----
;
; Quickdraw's oval-drawing calls are RAM-based. Load them.

LoadAgain    PushWord #4                    ;QuickDraw is tool set 4
              PushWord #$0100
              _LoadOneTool
              bcc ToolsLoaded

              cmp #VolNotFound
              beq DoMount
              sec
              ldx #$FE
              jsr PrepareToDie

DoMount      anop
              jsr MountBootDisk
              cmp #1
              beq LoadAgain

              sec
              rts

; The tools have been loaded.

ToolsLoaded  clc                            ;Clear the carry flag
              rts                            ; and return

              END

;*****
;
; Define the bowtie region, then draw it.
;
;*****

DrawTie      START
              using GlobalData

; This is the bowtie region's definition

              PushLong #0                    ;Allocate space
              _NewRgn

```

230 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

```

pla                                ;Read the handle into TieHandle
sta TieHandle
pla
sta TieHandle+2

_OpenRgn                          ;Start defining the region

PushWord #60                      ;Move pen to starting point
PushWord #45                      ; (45,60)
_MoveTo

PushWord #30                      ;Draw the left-hand triangle
PushWord #60                      ; moving clockwise
_LineTo
PushWord #30
PushWord #30
_LineTo
PushWord #60
PushWord #45
_LineTo

PushLong #KnotDef                 ;Draw the knot
_FrameOval

PushWord #70                      ;Draw the right-hand triangle
PushWord #45                      ; moving clockwise
_MoveTo
PushWord #100
PushWord #30
_LineTo
PushWord #100
PushWord #60
_LineTo
PushWord #70
PushWord #45
_LineTo

PushLong TieHandle                ;End the region definition
_CloseRgn

; Set up the pen and draw the bowtie

PushWord #1                      ;Set the pen color to red
_SetSolidPenPat

PushLong TieHandle                ; and paint the bowtie
_PaintRgn

rts
END

;*****
;
; Event Loop
;
; Keep the bowtie on the screen until user presses a key.
;
;*****

```



```
EventLoop      START
```

```
WaitHere       lda $E0C000
                bpl WaitHere
                lda $E0C010
                rts
```

```
END
```

```
*****
*
* Prepare To Die
*
* Dies if carry is set.
*
* Error number to display is in X register.
* Assumes that Miscellaneous Tools is active.
*
*****
```

```
PrepareToDie   START
                bcs RealDeath    ;Carry = 1?
                rts              ; No. Return to caller

RealDeath      phx              ; Yes. Goodbye, program.
                PushLong #DeathMsg
                _SysFailMgr

DeathMsg       str 'Could not handle error '

END
```

```
;*****
;
; Mount Boot Disk
;
; Tells the user to insert the disk that contains the RAM-based
; tools.
;
;*****
```

```
MountBootDisk START
```

```
    _Set_Prefix SetPrefixParams
    _Get_Prefix GetPrefixParams

    PushWord #0                ;Space for result
    PushWord #195              ;Column position for dialog box
    PushWord #30               ;Row position for dialog box
    PushLong #PromptStr        ;Prompt at top of dialog box
    PushLong #VolStr           ;Volume name string
    PushLong #OKStr            ;String in Button 1
    PushLong #CancelStr        ;String in Button 2
    _TLMountVolume

    pla                        ;Obtain the button number
    rts                        ; and return to caller
```

```

PromptStr      str 'Please insert the disk.'
VolStr         ds 16
OKStr          str 'OK'
CancelStr      str 'Shutdown'

GetPrefixParams dc i'7'
               dc i4'VolStr'
SetPrefixParams dc i'7'
               dc i4'BootStr'

BootStr        str '*/'

               END

```

Calculation Calls for Shapes

Earlier in this chapter, I described tool calls that move the pen to a specified location (MoveTo) or displace it a specified distance horizontally and vertically (Move). I also discussed LineTo and Line, which provide two ways to specify the end of a line. Similarly, QuickDraw also provides tool calls that move or displace shapes. These are handy for reproducing a shape at several places on the screen. To do this, you draw the shape, move it, then draw it again. The following are among the most useful calculation calls for shapes:

- SetRect and SetRectRgn change the corner coordinates of a rectangle or region to specified values, thereby moving it.
- OffsetRect, OffsetPoly, and OffsetRgn displace a rectangle, polygon, or region by specified horizontal and vertical values.
- CopyRgn copies the contents of one region into another.

A Color-Dithering Program

The DITHERS program in Example 7-4 employs OffsetRect to reproduce a rectangle called ColorBox at 16 positions across a 640-mode screen. DITHERS draws the leftmost box with a pen pattern of zeros (i.e., black), then enters a loop in which it displaces the box to the right, increments each pen pattern byte by 1, and redraws the box with the new pattern. The result is that each box has a different *dithered* color (see the earlier discussion of “Dithering in 640 Mode”). Box 0 is black, box 1 is blue, box 2 is gold, and so on.

Example 7-4

```

; DITHERS displays the 16 colors that can be obtained through dithering
; with the 640 mode's standard color table.

        absaddr on
        MCOPY Dithers.macros

Dithers      START
            using GlobalData

;-----
;
; Global equate used throughout the program.

ScreenMode   gequ $80                ;640 mode, no fill

            phk                      ;Set data bank to program
            plb                      ; bank to allow absolute addressing

            jsr InitStuff             ;Initialize everything
            bcs AllDone              ;Quit if initialization fails

            jsr ShowColors            ;Display the 16 color boxes

            jsr EventLoop             ;Wait for user to press a key

AllDone      anop                    ;All is done, shut down
            _QDS ShutDown             ;QuickDraw II
            _MT ShutDown              ;Miscellaneous Tools

            PushWord MyID             ;Discard the program's handle
            _DisposeAll

            PushWord MyID
            _MMS ShutDown             ;Memory Manager
            _TL ShutDown              ;Tool Locator

            _Quit QuitParams          ;Do a ProDOS Quit call
            brk $F0                  ;If it fails, break

        END

;*****
;
; Global Data
;
;*****

GlobalData   DATA

QuitParams   dc i4'0'                ;Return to caller
            dc i'$4000'              ;Make program restartable in memory

MyID         ds 2                    ;This will hold the program's i.d.

        END

```



```

;*****
;
; Show Colors
;
; Display the color boxes.
;
;*****

ShowColors START

        PushLong #PenPat           ;Set the pen pattern (i.e., color)
        _SetPenPat

        PushLong #ColorBox         ; and paint the leftmost box
        _PaintRect

; The following loop moves the color box 20 columns to the right,
; increments each pen pattern byte by one (to select the next
; dithered color), then paints the new color.

MoveBox  PushLong #ColorBox         ;Move the color box
        PushWord #20                ; 20 columns to the right
        PushWord #0
        _OffsetRect

BumpPat  ldx    #30                  ;Point to last 2 bytes in pattern
        clc                          ;Increment two bytes at a time
        lda    #$0101                ; by adding 1 to each of them
        adc    PenPat,x
        sta    PenPat,x
        dex
        dex                          ;Point to the preceding pair
        bpl    BumpPat                ; of bytes

        PushLong #PenPat             ;Set the pen color
        _SetPenPat

        PushLong #ColorBox           ; and paint the next box
        _PaintRect

        dec    BoxCount              ;Any more boxes to paint?
        bne    MoveBox

        rts                          ;If not, exit

BoxCount dc i'15'

ColorBox dc i'5'                    ;Initial rectangle extends from
        dc i'5'                      ; (5,5) to (20,20)
        dc i'20'
        dc i'20'

PenPat   dc h'00 00 00 00 00 00 00 00' ;Pen pattern (initially black)
        dc h'00 00 00 00 00 00 00 00'
        dc h'00 00 00 00 00 00 00 00'
        dc h'00 00 00 00 00 00 00 00'

END

```

```

;*****
;
; Event Loop
;
; Keep the boxes on the screen until user presses a key.
;
;*****

EventLoop      START

WaitHere       lda $E0C000
               bpl WaitHere
               lda $E0C010
               rts

               END

*****
*
* Prepare To Die
*
* Dies if carry is set.
*
* Error number to display is in X register.
* Assumes that Miscellaneous Tools is active.
*
*****

PrepareToDie   START
               bcs RealDeath      ;Carry = 1?
               rts                ; No. Return to caller

RealDeath      phx                ; Yes. Goodbye, program.
               PushLong #DeathMsg
               _SysFailMgr

DeathMsg       str 'Could not handle error '

               END

```

By simply changing the ScreenMode constant from \$80 to 0, you can make DITHERS display the 16 colors in the 320 mode's standard color table. As mentioned earlier, that's the advantage of using an equate directive to specify the display mode. Change the equate and you change the mode throughout the program. (Changing ScreenMode also lets you switch color tables. There's more about color tables at the end of this chapter.)

Mouse Pointer Tool Call

QuickDraw also provides a *ShowCursor* tool that puts a very important shape on the screen: the mouse pointer. To call it, simply enter **__ShowCursor**.

The mouse comes into play in Chapter 9, where I discuss windows and the “controls” that operate on them.

Text

As you will see later, the Apple IIGS has tool sets that let you display attractive boxed prompts with on-screen buttons (perhaps *OK* and *Cancel*) that the user can “press” to respond. However, sometimes you may simply want to put a message on the screen without concern as to how it looks. For example, you may want to show “Please wait . . .” while the program performs a lengthy operation. In situations like this, you can use QuickDraw calls to display the text.

Table 7-10 lists the most useful tools for setting the foreground (text) and background colors and “drawing” the text. QuickDraw always draws text at the current pen location and uses the current foreground and background colors.

For example, to display “Please wait . . .”, your program would contain:

```

                PushLong #WaitStr
                _DrawString
                . .
                . .
WaitStr      str  'Please wait...'

```

Pixel Images

Up to now, you have seen how to draw lines and shapes using QuickDraw’s pen. However, sometimes you may want to display a picture that contains more than just the predefined shapes. For example, you may want to show a landscape with grass, trees, and flowers beneath a cloudy blue sky.

You could certainly draw such a picture 1 pixel at a time, using the pen. But, needless to say, that would be a difficult, tedious, and time-consuming task. You’d probably see drawing calls in your sleep!

How much easier it would be just to tell QuickDraw which color belongs at each location — to give it a pixel-by-pixel color “map” of your picture — then make it display the entire picture or some portion of it. Well, you can do just that by defining the picture as a *pixel image*. Pixel images

Table 7-10

Color Calls	
<code>__SetBackColor</code>	Set the background color
Call with: <code>PushWord ColorNum</code>	
<code>__SetBackColor</code>	
Result: None	
Note: QuickDraw only uses the appropriate number of bits in <code>ColorNum</code> . In 320 mode, it uses 4 bits, so <code>ColorNum</code> can range from 0 to 15. In 640 mode, it uses 2 bits, so <code>ColorNum</code> can be 0, 1, 2, or 3 (for black, red, green, or white, respectively).	
<code>__SetForeColor</code>	Set the foreground color
Call with: <code>PushWord ColorNum</code>	
<code>__SetForeColor</code>	
Result: None	
Note: See note for <code>SetBackColor</code> .	

Drawing Calls	
<code>__DrawChar</code>	Display the specified character
Call with: <code>PushWord #'char' ;Character</code>	
<code>__DrawChar</code>	
Result: None	
<code>__DrawString</code>	Display the specified Pascal-style string
Call with: <code>PushLong StringPtr ;Pointer to string</code>	
<code>__DrawString</code>	
Note: <code>StringPtr</code> points to a Pascal-style string. It has the form:	
<code>dc il'count'</code>	
<code>dc 'string'</code>	
The “str” macro produces this format.	
<code>__DrawCString</code>	Display the specified C-style string
Call with: <code>PushLong StringPtr ;Pointer to string</code>	
<code>__DrawCString</code>	
Note: <code>StringPtr</code> points to a string of the form:	
<code>dc c'string'</code>	
<code>dc il'0'</code>	

are easier to understand if you know something about their counterparts (called “bit maps”) in the Macintosh.

Macintosh Bit Maps

The term *bit map* reflects the fact that on a black-and-white Macintosh screen, pixels can only be “on” (black) or “off” (white) — and you can

represent those two states with a single bit. Thus, a bit map is a collection of 1's and 0's: a 1 for each black pixel and a 0 for each white one. Let's look at an example.

Figure 7-12 shows an enlargement of the "pencil" icon displayed by the Macintosh drawing program, *MacPaint*. Its image area consists of 288 pixel positions: 18 rows by 16 columns. Hence, the bit map for this image would be 288 bits long.

You could, of course, construct the bit map as a series of 18-bit patterns, each 16 bits long. However, entering sequences of 1's and 0's is both time-consuming and error-prone. Since each row is 16 columns wide, it's easier to construct the bit map using 16-bit word values.

As you know, a word contains 4 hexadecimal digits, where each digit represents 4 bit positions. To determine the pencil's bit map entry for a particular row, you must mentally divide the row into 4 quarters and enter the hex digit each quarter represents.

For example, the second row of the pencil image (shown in Figure 7-13) has four black boxes, which means it will contribute four 1's to the bit map entry.

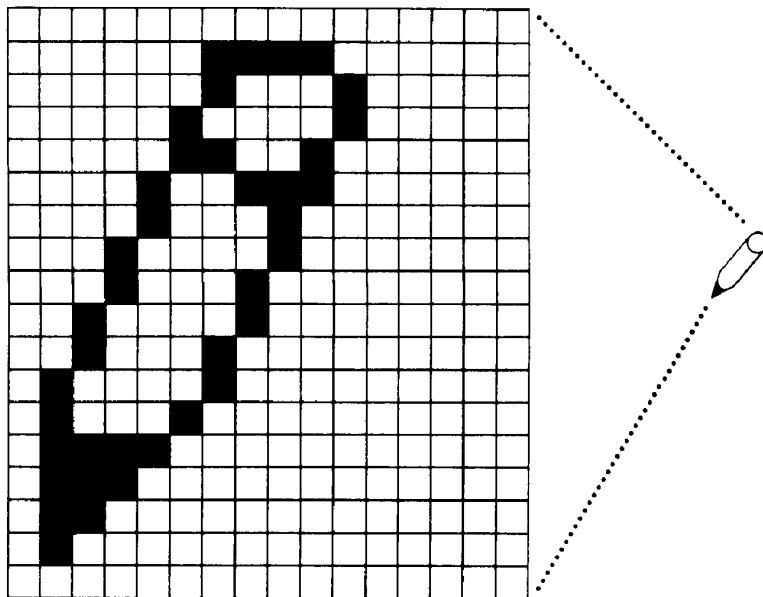


Figure 7-12

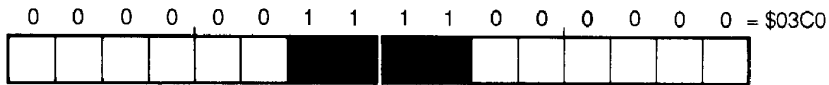


Figure 7-13

The first four pixel positions on this row are 0, so the first hex digit in the bit map entry is 0. The next four pixels have the pattern 0011, so the second hex digit is 3. The third group has the pattern 1100, so the third hex digit is C. The remaining pixels are 0, so the final hex digit is also 0. Combining the four digit produces the value \$03C0, so that's what belongs in the bit map entry for the second row.

After applying this pixels-to-digits conversion procedure to the other 17 rows, you win up with the entire image translated into 4-digit hex numbers, as shown in Figure 7-14. Thus, the bit map is stored in memory as a sequence of 18 4-digit numbers.

Assigning a single bit to each pixel position is reasonable for displaying

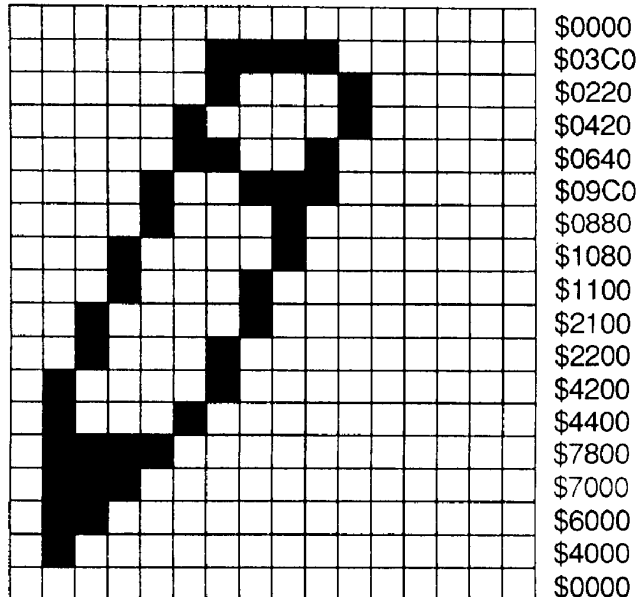


Figure 7-14

pictures on the Macintosh, because it only accomodates two colors: black (1) and white (0). But this approach is clearly inadequate for the Apple IIGS, because the IIGS must be able to display more than two colors at a pixel position. Specifically, it must display 16 colors in the 320 mode and four colors in the 640-mode. Obviously, then, a IIGS pixel image must be somewhat different from a Macintosh bit map.

Contents of a Pixel Image

What numeric values in a pixel image will produce colored pixels? As mentioned earlier, it takes a 4-bit pattern, or *nibble*, to select a color from the 16 entries in a 320-mode color table. As Table 7-2 shows, 0000 selects black, 0001 selects dark gray, and so on.

Let's return to my pencil image, with its 16-pixel rows. Bearing in mind that each pixel position must be represented by a 4-bit nibble (rather than a single bit), you need 16 nibbles — that is, 16 hex digits — to define a row. Assume that the pencil is to have a red body and black tip, and be displayed on a white background. In the 320-mode color table, red, black, and white have the pixel values \$7, \$0, and \$F, respectively. Thus, the first 8 pixels in the second row of the image are:

\$FFFF FF77

while the second eight pixels in this row are:

\$77FF FFFF

The hexadecimal values for the entire pencil image are shown in Figure 7-15.

The important point here is that *you must enter a color value for every pixel position in the image*. When displaying an image, QuickDraw does not apply the screen's current background pattern to unaffected pixels (as it does when drawing a boundary or frame with the pen), because there are no unaffected pixels!

Image Width

Since the pencil is only 10 pixels wide, you may wonder why the rows are 16 pixels wide. After all, the 5 columns of pixels to the right of it are extraneous; they only contain background data. The pencil image is 16 pixels wide because *QuickDraw can only work with images whose widths are multiples of 8 bytes*. Put another way, since there are 2 pixel numbers in

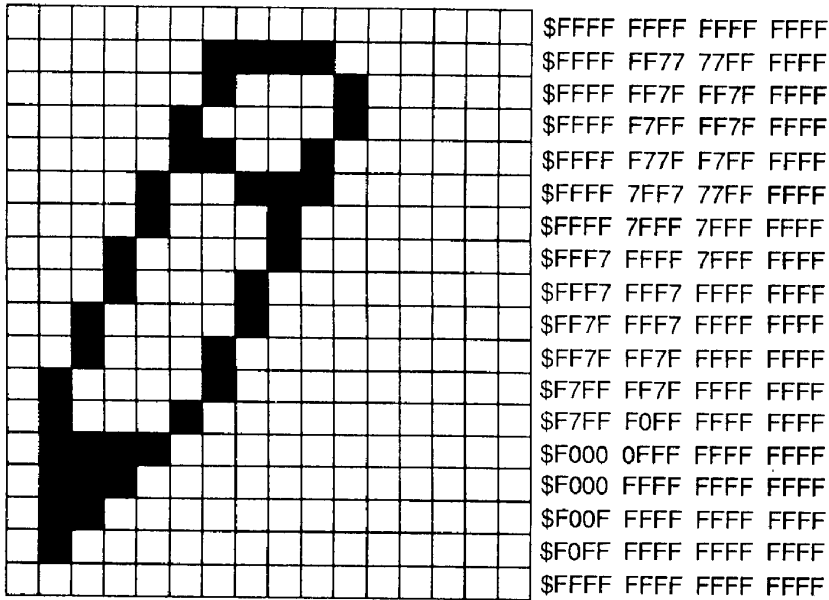


Figure 7-15

a byte, QuickDraw can only work with rows that are multiples of 16 pixels (that is, 16, 32, 48, and so on).

Fortunately, that doesn't mean that every pixel in the image is always displayed on the screen. You specify how much of the image is displayable by defining its "BoundsRect."

BoundsRect

The BoundsRect (short for boundary rectangle) is an imaginary frame that encloses the portion of an image that is to be available for display. For example, suppose you want to use only the first 12 columns of the pencil image — the pencil itself and the "background" column on its left and right. Further, suppose you want to situate the top left-hand corner of the displayable area at point (100,100) in the conceptual drawing space. To do this, you would assign the pencil image a BoundsRect that has the corner coordinates shown in Figure 7-16.

It's essential to realize that assigning a BoundsRect to a pixel image only positions it within the drawing space and scraps extraneous pixels. Assigning the BoundsRect does *not* display the image, nor does it even

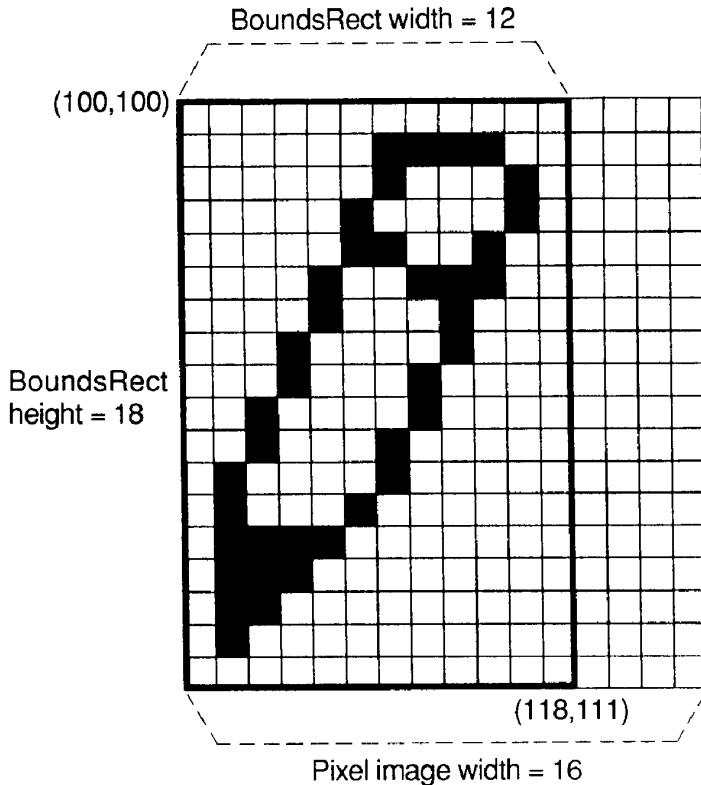


Figure 7-16

ensure that the image will ever be displayed. It simply makes the image *available* for display by QuickDraw.

To set up a BoundsRect, you must know something about the structure in which it dwells, the graphics port, or *GrafPort*.

The GrafPort

The GrafPort is a record of every parameter that relates to QuickDraw's current "personality" — how it's set to operate at the moment. The GrafPort keeps track of the current pen state, the foreground and background colors, the size and location of the BoundsRect, and so on. In other words, the GrafPort record oversees QuickDraw's *drawing environment*.

You cannot change an entry in the GrafPort record directly, but you do change one indirectly when you make a QuickDraw tool call that affects it. For example, SetPenSize and SetBackColor change the GrafPort's pen size and background color entries. In fact, whenever you make a QuickDraw tool call, QuickDraw checks the GrafPort to find out what pen size, background color, whatever, to use. QuickDraw tool calls are, in effect, calls to the GrafPort!

Entries in the GrafPort Record

Although it is not necessary to have expert knowledge of each and every detail of the GrafPort, you should have a general understanding of what it contains. Table 7-11 summarizes the entries in the GrafPort record.

Table 7-11. The GrafPort Record

PortInfo:	LocInfo
PortRect:	rect
ClipRgn:	handle
VisRgn:	handle
BkPat:	pattern
PnLoc:	point
PnSize:	point
PnMode:	integer
PnPat:	pattern
PnMask:	mask
PnVis:	integer
FontHandle:	handle
FontID:	longword
FontFlags:	integer
TxSize:	integer
TxFace:	style
TxMode:	integer
SpExtra:	fixed
ChExtra:	fixed
FGColor:	integer
BGColor:	integer
PicSave:	handle
RgnSave:	handle
PolySave:	handle
GrafProcs:	pointer
ArcRot:	integer
UserField:	long word
SysField:	long word

This first field, *PortInfo*, contains a data structure called *LocInfo* that is arranged as follows:

```
PortSCB: word
PointerToPixelImage: pointer
ImageWidth: word
BoundsRect: rect
```

Here, *PortSCB* is a word value that contains the *GrafPort*'s scan line control byte (SCB). The *PortSCB* does for the port what the master SCB does for QuickDraw overall (see Figure 7-5 for the SCB's format); that is, it specifies the graphics mode (320 or 640) and the color table.

The last three items in *LocInfo* incorporate your program's pixel data into the *BoundsRect*. *PointerToPixelImage* points to the data table for the pixel image, and *ImageWidth* specifies the width of the image's rows in bytes. Together, these two parameters tell QuickDraw where to find the image's data table and how to subdivide the long sequence of numbers it contains.

BoundsRect, the last *LocInfo* parameter, provides the corner coordinates of the image's boundary rectangle. These coordinates tell QuickDraw what portion of the conceptual drawing space to assign to the image. Further, the width of the *BoundsRect* tells QuickDraw how much of the image is available for display. (Remember, you can discard extraneous pixels by making the *BoundsRect* narrower than the pixel image table.)

The *PortRect*, the second item in the *GrafPort* record, is just as important as the *BoundsRect*. For an image or any part of it to actually be displayed — that is, to be visible on the screen — it must lie within *PortRect*'s border. Thus, the portion of an image that appears on the screen is the part that's inside both the *BoundsRect* and the *PortRect*. In other words, it is the area where these rectangles overlap or intersect.

To draw an analogy (and risk overstating my point), the *BoundsRect* is like a window in a house or office; it gives you the opportunity to see part of a much larger picture — the world outside. The *PortRect* is like a rectangular telescope through which you can look out the *BoundsRect* window.

How much you actually see through the telescope depends on its field of vision and whether it is pointing directly at the window. That is, what appears on the screen depends on the size of the *PortRect* and its orientation relative to the *BoundsRect*. If you still find this concept difficult to understand, Figure 7-17 should help make it clearer.

ClipRgn, short for clip region, lets you “lock” a portion of the *PortRect*

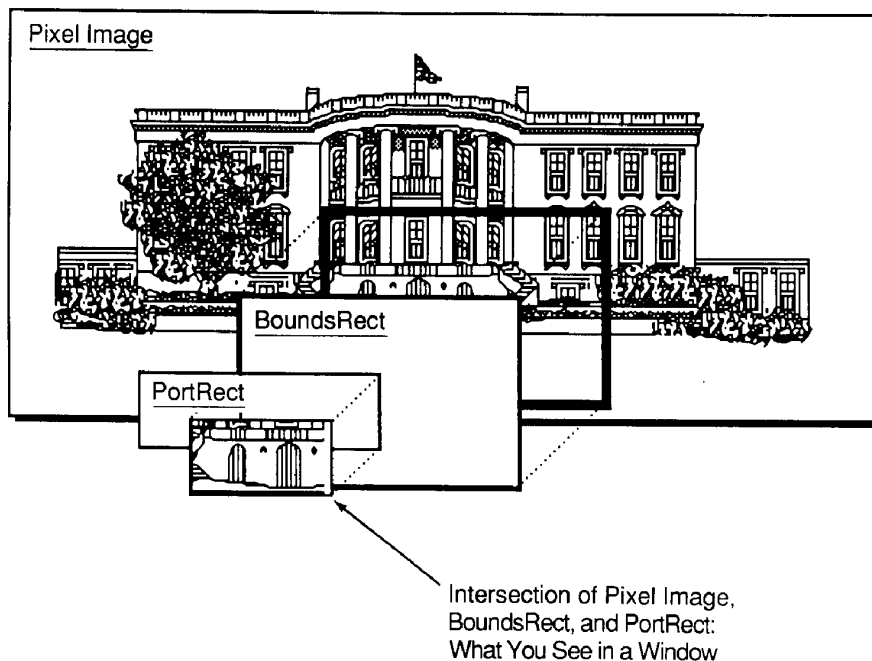


Figure 7-17

so that its pixels do not change. This can be handy in graphics programs that contain animation.

For example, imagine a game program that shows characters walking along a city street that has a large mailbox in the foreground. If the program has designated the mailbox area as a clip region, characters passing behind it would appear partially invisible. Without clipping, the program would have to change character pixels that are obscured by the mailbox; with a *ClipRgn* active, the obscured pixels are blocked out automatically. Although they are still within the intersection of the *BoundsRect* and *PortRect* — the normal conditions for being displayed — the fact that they are also within the clip region prevents them from appearing on the screen.

Note that *ClipRgn* is a region. It can be rectangular, like the *PortRect* and *BoundsRect*, but it need not be. As with regions you define for drawing with *QuickDraw*, the *ClipRgn* assumes whatever shape you give it.

VisRgn is used by the Memory Manager to display windows that overlap.

BkPat establishes the *GrafPort*'s background pattern. It can be changed with the calls *SetBackPat* and *SetSolidBackPat*.

You already understand *PnSize*, *PnMode*, *PnPat*, and *PnMask*. They are pen state attributes.

Sometimes you may want to deactivate the pen entirely, to disregard drawing calls in a certain part of the program. *PnVis* controls this condition. Call *HidePen* to deactivate the pen and *ShowPen* to reinstate it.

The next eight items, *FontHandle* through *ChExtra*, have to do with text and the font used to display it. *FGColor* and *BGColor* determine the foreground and background colors for text. They are affected by *SetForeColor* and *SetBackColor*.

You can generally ignore the remaining items in the record. They're used by system programs, not application programs.

Multiple GrafPorts

QuickDraw is not the only tool set that sets up a GrafPort; the *Window Manager* also sets one up whenever you open a new window. This makes it easy to work with multiple windows, because each has its own GrafPort — that is, its own unique drawing environment. Thus, when you switch between windows that overlap on the screen, the Window Manager activates the GrafPort for the window you want to work with. It thereby saves you the trouble of remembering the GrafPort settings for each individual window. Windows and the GrafPorts associated with them are discussed in Chapter 9.

Table 7-12 lists two tool calls you may need when working with multiple windows. The first, *GetPort*, returns a pointer to the current GrafPort. You use *GetPort* when you want to do something in another port (window), but need to remember this port so you can switch back to it later. *SetPort* is the call you use to switch ports.

Table 7-12

<hr/>		
<i>__GetPort</i>		Get handle of the current GrafPort
Call with:	PushLong #0 ;Space for result (handle)	
	<i>__GetPort</i>	
Result:	Pointer to port (long word)	
<i>__SetPort</i>		Switch to the specified GrafPort
Call with:	PushLong <i>PortPtr</i> ;Pointer to port	
	<i>__SetPort</i>	
Result:	None	
<hr/>		

Displaying a Pixel Image

The preceding excursion into the world of the GrafPort gave me the opportunity to introduce the PortRect and describe its relationship to the BoundsRect. Let's take stock of where we are.

In the "Pixel Images" section, you learned how to set up a pixel image using a table of nibble-size, 320-mode color values. Since QuickDraw requires rows in images to be multiples of 8 bytes (or 16 pixels) wide, I introduced the BoundsRect, which lets you discard extraneous pixels. Because a BoundsRect only establishes a coordinate system for an image, however, it was necessary to introduce the PortRect. The intersection of the PortRect and BoundsRect determines what portion of the pixel image actually appears on the screen.

With all this theory presented, it's time to create a program that actually displays a pixel image. The pencil I have used all along is as good as any, so I'll stick with it.

Tool Calls to Display Pixel Images

To write a program that displays a pixel image, you must know which tool calls specify the image and set up the BoundsRect and PortRect. PPToPort, shown in Table 7-13, is the call you normally use to display an image.

The ScrollRect call moves a pixel image by specified horizontal and vertical displacements (dHoriz and dVert) within a rectangle to which RectPtr points. The displacements are signed numbers (integers), with positive values moving the image to the right or downward, respectively. The rectangle simply establishes the boundaries in which the image can move and still be visible. In other words, with ScrollRect, for an image to appear on the screen, it must lie within the intersection of the BoundsRect, PortRect, and the so-called scroll-area rectangle to which RectPtr points.

ScrollRect's calling sequence doesn't indicate *which* image it will move, or "scroll." That's because it obtains this information from the GrafPort automatically. That is, it always scrolls the image that is within the current BoundsRect and PortRect.

The final input to ScrollRect — UpRgnHandle — is simply the handle of a region in memory that QuickDraw will use as working storage for the move operation. ScrollRect does not create this region; you must provide it with a preceding NewRgn call. Since ScrollRect is primarily used to animate images, I will discuss it further in an upcoming "Animation" section.

Since PPToPort assumes you have already defined the PortRect (or accept the default, the entire screen), three PortRect calls are also listed.

Table 7-13

__PPToPort		Display a pixel image
Call with:	PushLong <i>LocInfoPtr</i> ;Pointer to parameter block PushLong <i>ImageRectPtr</i> ;Pointer to image rectangle PushWord <i>ScreenColumn</i> ;Screen column and PushWord <i>ScreenRow</i> ; row PushWord <i>PenMode</i> ;Pen transfer mode __PPToPort	
Result:	None	
Note:	LocInfoPtr points to: PortSCB (word) PointerToPixelImage (pointer) ImageWidth (word) BoundsRect (rect)	
__ScrollRect		Move a pixel image within a specified rectangle
Call with:	PushLong <i>RectPtr</i> ;Pointer to scroll area rect. PushWord <i>dHoriz</i> ;Horizontal displacement PushWord <i>dVert</i> ;Vertical displacement PushLong <i>UpRgnHandle</i> Handle of update region __ScrollRect	
Result:	None	
Note:	See text.	
__SetOrigin		Start PortRect at the specified point
Call with:	PushWord <i>Column</i> ;Column and PushWord <i>Row</i> ; row of top left corner __SetOrigin	
Result:	None	
__GetPortRect		Get the current PortRect coordinates
Call with:	PushLong <i>RectPtr</i> ;Pointer to buffer __GetPortRect	
Result:	None	
Note:	RectPtr points to: ds 2 ;Row and ds 2 ; column of top left corner ds 2 ;Row and ds 2 ; column of bottom right corner	
__SetPortRect		Set the PortRect coordinates
Call with:	PushLong <i>RectPtr</i> ;Pointer to PortRect coords. __SetPortRect	
Result:	None	
Note:	See note for GetPortRect.	

Note the similar calls `SetOrigin` and `SetPortRect`, which both change the PortRect. They differ in that `SetOrigin` only moves the upper left-hand corner of the PortRect (and leaves the bottom right-hand corner unchanged), whereas `SetPortrect` redefines it entirely.

Pencil Display Program

Example 7-5 lists a program called PENCIL that displays the pencil image (called PclImg here) on a 320 mode screen. In fact, PENCIL displays the pencil *twice*, at locations (0,0) and (50,50), on a light blue background. Here, the BoundsRect encloses only the first 12 columns (0-11) of the image, because columns 12 through 15 are extraneous.

Animation

In some applications you may want to move graphics objects (shapes or pixel images) on the screen, or *animate* them. Producing animation isn't as difficult as you might expect. In fact, it requires only five steps:

1. Display the object.
2. Wait a period of time, so the object is visible.
3. Erase the object.
4. Move to where you want the object to appear next.
5. Repeat the process, starting at Step 1.

Example 7-5

```
; PENCIL displays the pixel image of a pencil icon in 320 mode.

      absaddr on
      MCOPY Pencil.macros

Pencil      START
            using GlobalData

;-----
;
; Global equate used throughout the program.

ScreenMode      gequ 0                      ;320 mode, no fill

               phk                          ;Set data bank to program
               plb                          ; bank to allow absolute addressing

               jsr InitStuff                ;Initialize everything
               bcs AllDone                  ;Quit if initialization fails

               jsr ShowPencil               ;Display the pencil
```

```

                                jsr EventLoop                ;Wait for user to press a key

AllDone      anop                    ;All is done, shut down
              _QDShutDown            ;QuickDraw II
              _MTShutDown            ;Miscellaneous Tools

              PushWord MyID          ;Discard the program's handle
              _DisposeAll

              PushWord MyID
              _MMShutDown            ;Memory Manager
              _TLShutDown            ;Tool Locator

              _Quit QuitParams       ;Do a ProDOS Quit call
              brk $F0                ;If it fails, break

              END

;*****
;
; Global Data
;
;*****

GlobalData   DATA

QuitParams   dc i4'0'                ;Return to caller
              dc i'$4000'            ;Make program restartable in memory

MyID         ds 2                    ;This will hold the program's i.d.

              END

;*****
;
; InitStuff
;
; Initializes tool sets and gets space in bank 0 for use as
; direct page by tools that need it.
;
;*****

InitStuff    START
              using GlobalData

; Initialize the Tool Locator, Memory Manager, and Miscellaneous
; Tools.

              _TLStartup

              PushWord #0
              _MMStartup

              pla                    ;Memory Manager returns program's ID
              sta MyID

              _MTStartup
              ldx #3
              jsr PrepareToDie

```

252 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

```
; Get some space for the direct page we need. QuickDraw needs
; three pages.

    PushLong #0                ;Space for handle
    PushLong #$300            ;Three pages for QuickDraw
    PushWord MyID              ;Owner
    PushWord #$C005           ;Locked, fixed, fixed bank
    PushLong #0                ;Location
    _NewHandle
    ldx #$FF
    jsr PrepareToDie

    pla                        ;Read handle and store in dp
    sta 0
    pla
    sta 2

    lda [0]                    ;Get dp location from handle

; Initialize QuickDraw

    pha                        ;Use dp obtained from handle
    PushWord #ScreenMode       ;Mode = 640
    PushWord #0                ;Use entire screen
    PushWord MyID
    _QDStartup
    ldx #4
    jsr PrepareToDie

    clc                        ;Clear the carry flag
    rts                        ; and return

END

;*****
;
; Display the pencil at two places on the screen: (0,0) and (50,50).
;
;*****

ShowPencil START
    using PencilData

    PushWord #$BBBB            ;Set background to light blue
    _ClearScreen

    PushLong #PclImg           ;Store pencil image pointer
    pla
    sta Ptr2PI                 ; in LocInfo
    pla
    sta Ptr2PI+2

    PushLong #LocInfo           ;Pointer to parameter block
    PushLong #PencilRect       ;Pointer to source rectangle
    PushWord #0                 ;Start at column 0
    PushWord #0                 ; and row 0
    PushWord #0                 ;Use normal pen mode
    _PPToPort

    PushLong #LocInfo           ;Pointer to parameter block
    PushLong #PencilRect       ;Pointer to source rectangle
```

```

        PushWord #50                ;Start at column 50
        PushWord #50                ; and row 50
        PushWord #0                 ;Use normal pen mode
        _PToPort

        rts
    END

;*****
;
; Event Loop
;
; Keep the pencil on the screen until user presses a key.
;
;*****

EventLoop    START

WaitHere     lda $E0C000
             bpl WaitHere
             lda $E0C010
             rts

             END

;*****
;
; Pencil Data
;
; Pixel image and definition of the image.
;
;*****

PencilData DATA

PclImg      dc h'ffff ffff ffff ffff'      ;Row 1
             dc h'ffff ff77 77ff ffff'      ;Row 2
             dc h'ffff ff7f ff7f ffff'      ;Row 3
             dc h'ffff f7ff ff7f ffff'      ;Row 4
             dc h'ffff f77f f7ff ffff'      ;Row 5
             dc h'ffff 7ff7 77ff ffff'      ;Row 6
             dc h'ffff 7fff 7fff ffff'      ;Row 7
             dc h'ffff ffff 7fff ffff'      ;Row 8
             dc h'ffff7 fff7 ffff ffff'      ;Row 9
             dc h'fff7f fff7 ffff ffff'      ;Row 10
             dc h'fff7f fff7 ffff ffff'      ;Row 11
             dc h'f7ff fff7 ffff ffff'      ;Row 12
             dc h'f7ff f0ff ffff ffff'      ;Row 13
             dc h'f000 0fff ffff ffff'      ;Row 14
             dc h'f000 ffff ffff ffff'      ;Row 15
             dc h'f00f ffff ffff ffff'      ;Row 16
             dc h'f0ff ffff ffff ffff'      ;Row 17
             dc h'ffff ffff ffff ffff'      ;Row 18

PencilRect  dc i'0,0,17,15'      ;Coordinates of pencil image

; This is the LocInfo to be sent to the GrafPort

```

```

LocInfo  anop
PortSCB  dc i'0'                                ;SCB
Ptr2PI   ds 4                                    ;Pointer to image
Width    dc i'8'                                ;Width of image (bytes)
BoundsRect dc i'0,0,17,11'                      ;BoundsRect

                END

*****
*
* Prepare To Die
*
* Dies if carry is set.
*
* Error number to display is in X register.
* Assumes that Miscellaneous Tools is active.
*
*****

PrepareToDie  START
               bcs RealDeath      ;Carry = 1?
               rts                ; No. Return to caller

RealDeath     phx                  ; Yes. Goodbye, program.
               PushLong #DeathMsg
               _SysFailMgr

DeathMsg      str 'Could not handle error '

                END

```

Generating a wait interval or delay, as prescribed in Step 2, takes some non-QuickDraw tool calls that I'll describe shortly. The instructions and tool calls you need for the other steps depend on what kind of objects you are animating — shapes or pixel images.

Animating Shapes

You can probably guess which calls you need to animate a QuickDraw shape. You need a “Frame,” “Paint,” or “Fill” type call to draw it, an “Erase” type call to erase it, and a “Set” or “Offset” type call to move it to the next display position. For example, you need `EraseRgn` to erase a region and either `SetRectRgn` or `OffsetRgn` to move it.

Animating Pixel Images

Believe it or not, pixel images are somewhat easier to animate than regular shapes, because you can use a single tool call to erase the image and move

it to its next position. The tool that does these jobs is *ScrollRect* (summarized in Table 7-13). *ScrollRect* not only erases the image and moves it, but it also updates the image's location.

Example 7-6 lists a program called ANIMATE that moves the pencil icon across a 320 mode screen. ANIMATE is simply a modified version of PENCIL (Example 7-5) in which the *MoveImg* segment (called *ShowPencil* before) contains a *NewRgn* call to allocate a scroll area in memory for the image and a *ScrollRect* call to do the scrolling.

Example 7-6

```
; ANIMATE moves a pencil icon across a 320 mode screen.

      absaddr on
      MCOPY Animate.macros

Animate      START
             using GlobalData

;-----
;
; Global equate used throughout the program.

ScreenMode   gequ 0                      ;320 mode, no fill

             phk                          ;Set data bank to program
             plb                          ; bank to allow absolute addressing

             jsr InitStuff                 ;Initialize everything
             bcs AllDone                  ;Quit if initialization fails

             jsr MoveImg                   ;Do the animation

             jsr EventLoop                 ;Wait for user to press a key

AllDone      anop                          ;All is done, shut down
             _QDShutDown                   ;QuickDraw II
             _MTShutDown                    ;Miscellaneous Tools

             PushWord MyID                  ;Discard the program's handles
             _DisposeAll

             PushWord MyID
             _MMShutdown                    ;Memory Manager
             _TLShutdown                    ;Tool Locator

             _Quit QuitParams               ;Do a ProDOS Quit call
             brk $F0                        ;If it fails, break

END
```

256 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

```
;*****
;
; Global Data
;
;*****

GlobalData    DATA

ImgHandle     ds 4                      ;This will hold image's handle

QuitParams    dc i4'0'                  ;Return to caller
               dc i'$4000'              ;Make program restartable in memory

MyID          ds 2                      ;This will hold the program's i.d.

               END

;*****
;
; InitStuff
;
; Initializes tool sets and gets space in bank 0 for use as
; direct page by tools that need it.
;
;*****

InitStuff     START
               using GlobalData

; Initialize the Tool Locator, Memory Manager, and Miscellaneous
; Tools.

               _TLStartup

               PushWord #0
               _MMStartup

               pla                      ;Memory Manager returns program's ID
               sta MyID

               _MTStartup
               ldx #3
               jsr PrepareToDie

; Get some space for the direct page we need. QuickDraw needs
; three pages.

               PushLong #0              ;Space for handle
               PushLong #$300           ;Three pages for QuickDraw
               PushWord MyID            ;Owner
               PushWord #$C005          ;Locked, fixed, fixed bank
               PushLong #0              ;Location
               _NewHandle
               ldx #$FF
               jsr PrepareToDie

               pla                      ;Read handle and store in dp
               sta 0
               pla
               sta 2
```

```

        lda [0]                ;Get dp location from handle
; Initialize QuickDraw

        pha                    ;Use dp obtained from handle
        PushWord #ScreenMode    ;Mode = 640
        PushWord #0             ;Use entire screen
        PushWord MyID
        _QDStartup
        ldx #4
        jsr PrepareToDie

        clc                    ;Clear the carry flag
        rts                    ; and return

        END

;*****
;
; Move Image
;
; Display the pencil, then move it.
;
;*****

MoveImg START
using GlobalData
using PencilData

        PushWord #$BBBB        ;Set background to light blue
        _ClearScreen

        PushWord #$B           ; and BackPat to same color
        _SetSolidBackPat

        PushLong #PclImg       ;Store pencil image pointer
        pla
        sta Ptr2PI             ; in LocInfo
        pla
        sta Ptr2PI+2

; Display the pencil at its starting position (0,0).

        PushLong #LocInfo      ;Pointer to parameter block
        PushLong #PencilRect   ;Pointer to source rectangle
        PushWord #0            ;Start at column 0
        PushWord #0            ; and row 0
        PushWord #0            ;Use normal pen mode
        _PPToPort

        PushLong #0            ;Set up a region for the image
        _NewRgn
        pla                    ;Read the handle into ImgHandle
        sta ImgHandle
        pla
        sta ImgHandle+2

; Move the image to the right 60 times, advancing 5 point positions
; each time.

```

258 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

```

MoveAgain anop
    PushLong #ScrollRect        ;Pointer to scroll rectangle
    PushWord #5                 ;Move the image 5 columns
    PushWord #0                 ; to the right
    PushLong ImgHandle          ;Specify its region's handle
    _ScrollRect

    dec MoveCount
    bne MoveAgain

    rts

MoveCount dc i'60'              ;Move count variable

END

;*****
;
; Event Loop
;
; Keep the pencil on the screen until user presses a key.
;
;*****

EventLoop    START

WaitHere     lda $E0C000
             bpl WaitHere
             lda $E0C010
             rts

             END

;*****
;
; Pencil Data
;
; Pixel image and definition of the image.
;
;*****

PencilData DATA

PclImg      dc h'ffff ffff ffff ffff'        ;Row 1
             dc h'ffff ff77 77ff ffff'        ;Row 2
             dc h'ffff ff7f ff7f ffff'        ;Row 3
             dc h'ffff f7ff ff7f ffff'        ;Row 4
             dc h'ffff f77f f7ff ffff'        ;Row 5
             dc h'ffff 7fff 77ff ffff'        ;Row 6
             dc h'ffff 7fff 7fff ffff'        ;Row 7
             dc h'fff7 ffff 7fff ffff'        ;Row 8
             dc h'fff7 ffff ffff ffff'        ;Row 9
             dc h'fff7 ffff ffff ffff'        ;Row 10
             dc h'fff7 ff7f ffff ffff'        ;Row 11
             dc h'f7ff ff7f ffff ffff'        ;Row 12
             dc h'f7ff f0ff ffff ffff'        ;Row 13
             dc h'f000 0fff ffff ffff'        ;Row 14
             dc h'f000 ffff ffff ffff'        ;Row 15

```

```

        dc h'f00f ffff ffff ffff'      ;Row 16
        dc h'f0ff ffff ffff ffff'      ;Row 17
        dc h'ffff ffff ffff ffff'      ;Row 18

PencilRect dc i'0,0,17,15'      ;Coordinates of pencil image
ScrollRect dc i'0,0,17,320'     ;Scroll rectangle

; This is the LocInfo to be sent to the GrafPort

LocInfo  anop
PortSCB  dc i'0'                ;SCB
Ptr2PI   ds 4                   ;Pointer to image
Width    dc i'8'                ;Width of image (bytes)
BoundsRect dc i'0,0,17,11'      ;BoundsRect

END

*****
*
* Prepare To Die
*
* Dies if carry is set.
*
* Error number to display is in X register.
* Assumes that Miscellaneous Tools is active.
*
*****

PrepareToDie  START
              bcs RealDeath      ;Carry = 1?
              rts                ; No. Return to caller

RealDeath     phx                ; Yes. Goodbye, program.
              PushLong #DeathMsg
              _SysFailMgr

DeathMsg      str 'Could not handle error '

END

```

Note that the PushWord #5 input to ScrollRect makes it move the pencil five columns to the right each time. I arrived at this number through some trial and error. At the outset, I tried moving the image one column at a time. The image moved fairly smoothly, so the animation looked fine, but it was tortuously slow. The pencil took about 35 *seconds* to cross the screen!

Next I tried moving the image an entire width (18 points) at a time. It then took about 3 seconds to cross the screen, but the movement was very jumpy. After more experimenting, I finally settled on moving five points at a time. That seems close to optimal for both speed and appearance.

Time and Date Operations

The delay interval needed in an animation routine to make a displayed object visible is just one of many operations that pertain to the time or date. For instance, you may want to display the elapsed time or the remaining time for a game that involves specific intervals. Obvious examples are football and basketball simulations; each game uses a clock to show the time remaining in a quarter or period. Similarly, a computerized chess game may include a timer that requires the player to move within some set period. Educational programs often include timers for the same purpose — to set a limit on the amount of time a student can use to solve a set of problems.

These examples all involve intervals of time. You may also want to use the current time and date directly, to “time stamp” a document.

These are just a few situations where having access to the time and date is handy, and you can probably think of many others. At any rate, be thankful for the battery backed-up clock that’s built into the Apple IIGS; it’s the key to performing just about any time- or date-related job you want to do.

Tool Calls for Reading the Time and Date

The Apple IIGS Miscellaneous Tools set contains tools that return the current time and date, as stored in the battery backed-up RAM. Built into ROM, Miscellaneous Tools contains what you might expect: tools that don’t fall into any category that’s broad enough to warrant its own tool set. Included are tools that interface with the mouse, read and change parameters in the battery backed-up RAM (the parameters used by the Control Panel), and do a variety of other jobs.

However, the only tools that are of interest here are *ReadTimeHex* and *ReadAsciiTime*, which return the time and date on the stack or as a text string, respectively. Table 7-14 summarizes these calls and the start-up and shutdown calls for the Miscellaneous Tools.

ReadTimeHex returns 4 words, with a different parameter in each byte. Here is a description of the words, listed in the order you pull them off the stack:

- The first word contains the Minute (high byte) and Second values.
- The second word contains the Year (minus 1900) and Hour (0-23) values.
- The third word contains the Month and Day values. The Month can range from 0 to 11, where 0 is January.

Table 7-14

<u>MTStartup</u>		Start the Miscellaneous Tools
Call with: <u>MTStartup</u>		
Result: None		
<u>MTShutDown</u>		Shut down the Miscellaneous Tools
Call with: <u>MTShutDown</u>		
Result: None		
<u>ReadTimeHex</u>		Read the time and date in hex format
Call with: PushWord #0		;Space for 4 result words
PushWord #0		
PushWord #0		
PushWord #0		
<u>ReadTimeHex</u>		
Results: Four word values; see text.		
<u>ReadAsciiTime</u>		Read the time and date in ASCII format
Call with: PushLong <i>StringPtr</i>		;Pointer to string buffer
<u>ReadAsciiTime</u>		
Result: None		
Note: Returns 20 characters, with the high-order bit of each character set to one.		
See text for string formats.		

- The high byte of the fourth word contains the Day of the Week (0-6, where 0 is Sunday); its low byte is unused and contains 0.

Note that Read Time Hex can only produce the time to the nearest second; it doesn't report hundredths or even tenths of a second, so it wouldn't be very useful as a stopwatch. To get that degree of accuracy, you would have to use another Miscellaneous Tools call, *GetTick*. The so-called tick count that *GetTick* returns gets incremented 60 times a second.

The other date-and-time call, *ReadAsciiTime*, returns the date and time as a 20-character ASCII text string. The format of this string depends on which date and time format is active in the Control Panel's "Clock" options. The possible formats are as follows (with the first one being the default):

Date Format	Time Format	ReadAsciiTime String		
MM/DD/YY	AM-PM	mm/dd/yy	HH:MM:SS	AM or PM
DD/MM/YY	AM-PM	dd/mm/yy	HH:MM:SS	AM or PM
YY/MM/DD	AM-PM	yy/mm/dd	HH:MM:SS	AM or PM
MM/DD/YY	24 Hour	mm/dd/yy	HH:MM:SS	
DD/MM/YY	24 Hour	dd/mm/yy	HH:MM:SS	
YY/MM/DD	24 Hour	yy/mm/dd	HH:MM:SS	

Here, the “24 Hour” time format indicates international time, with hours ranging from 0 (midnight) to 23 (11:00 P.M.).

Standard ASCII characters are 8-bit values in which the high-order bit is set to 0 (see Appendix B). However, ReadAsciiTime’s characters have the high bit set to 1! This means that if you want to display the string, you have to strip off the high bit from each character byte. Example 7-7 shows a sub-routine called GetTime that does this.

Earlier, I promised to talk about generating delays, so here it is.

Generating Delays

Programs that generate delays usually do the following:

1. Read the current time.
2. Add selected increments to the time, to produce a “target” time — the time at which the delay should end.

Example 7-7

```
GetTime  START

          PushLong #TimeString      ;Read the time and date
          _ReadAsciiTime

; ReadAsciiTime returns characters with the high bit set to 1.
; I must make these bits 0 to display the string.

NextWord  ldx #18                    ;Strip off the high bit
          lda TimeString,x          ; 2 bytes at a time
          and #$7F7F
          sta TimeString,x
          dex
          dex
          bpl NextWord

          PushWord #10               ;Move the pen to (10,10)
          PushWord #10
          _MoveTo

          PushLong #TimeString      ; and display the string
          _DrawCString

          rts

TimeString ds 20
          dc il'0'

END
```


3. Adjust the target time so that hours don't exceed 23 and minutes and seconds don't exceed 59. This involves carrying any excess to the next higher unit.
4. Read the time repeatedly until the current time matches or exceeds the target time.

Figure 7-17 shows a flowchart for producing a time delay based on user-specified increments which are added to the minute and second values. (I assume you won't want to delay for hours at a time, although you could.)

So now the job at hand is to convert the flowchart into a usable program. In essence, the program should consist of a couple of `ReadTimeHex` calls separating assembly language instructions that add the increments to the initial time and then adjust the target time so that it contains legal values. The subroutine in Example 7-8, called `DELAY`, shows one way of doing these jobs. `DELAY` obtains the duration of the delay from `A` (seconds) and `Y` (minutes).

Although the listing is longer than one might expect for such a simple job, most of the bulk results from the fact that `ReadTimeHex` packs its time and date values into individual bytes of a word. Since byte data is awkward to work with in 16-bit native mode, the results were unpacked and stored in word variables. Then, after the target time was calculated, the words were repacked into `ReadTimeHex`'s byte-oriented format.

Because `DELAY` is based on `ReadTimeHex`, the delays it produces may not be very accurate. For example, if you request a 3-second delay when the current time is 8:29:00:90 (that is, 90/100 of a second past 8:29), `ReadTimeHex` ignores the fraction and returns 8:29. In response, `DELAY` adds 3 seconds and sets the target time to 8:32. However, 1/10 of a second later, the time changes to 8:30, which means you actually receive a delay of 2.10 seconds — far short of what you want.

The moral of the story is: *DELAY's accuracy may be off by nearly a second.* If this is unacceptable, you may want to resort to `Get Tick` is accurate to 1/60 of a second.

To use `DELAY` in a program, simply call it with a sequence of the form:

```
lda seconds      ;Number of seconds to wait
ldy minutes      ;Number of minutes to wait
jsr Delay        ;Start waiting
```

and put a *copy delay.src* statement at the end of your program.

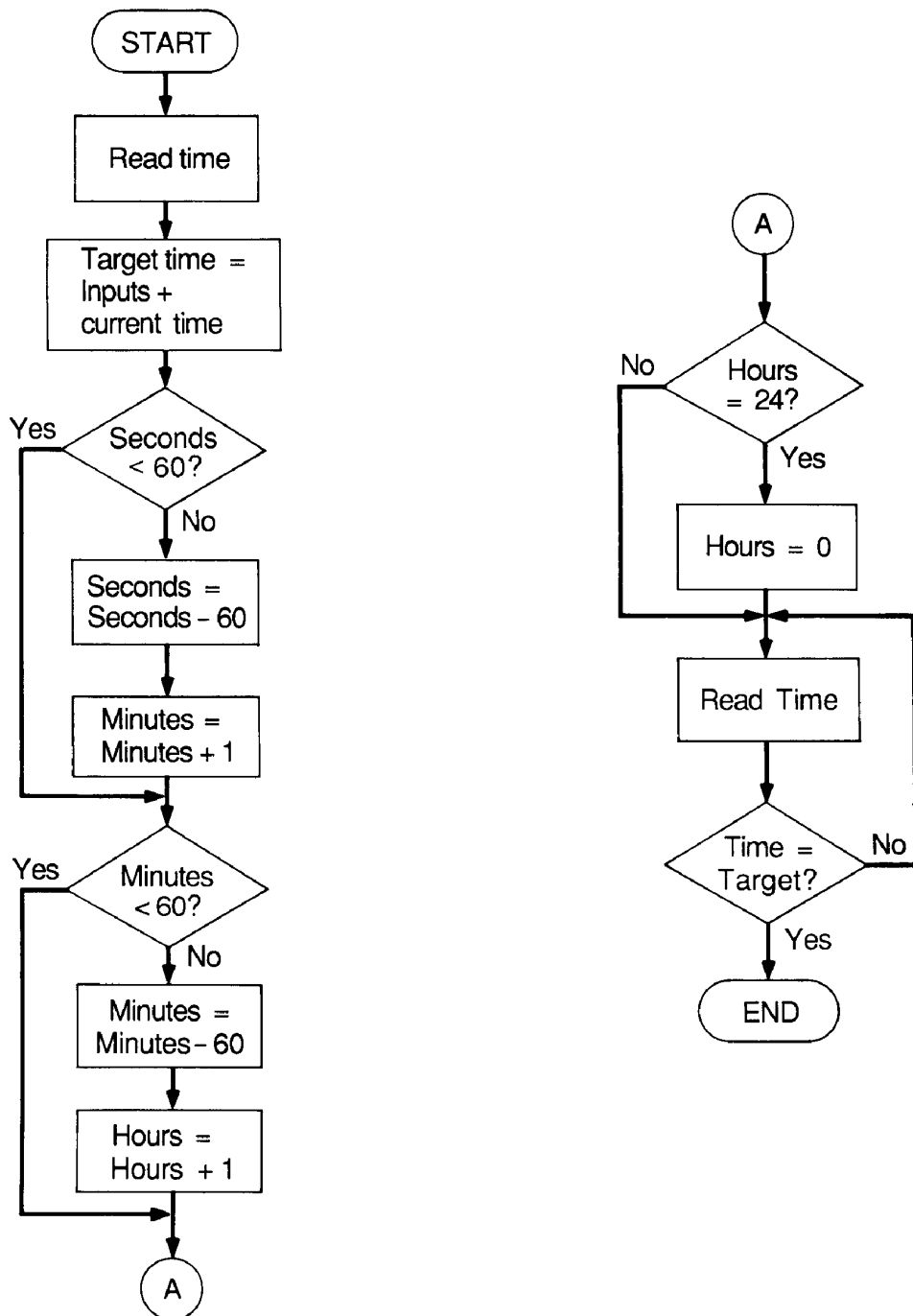


Figure 7-17

Example 7-8

```

;*****
;
; DELAY generates a delay interval based on a seconds count in the
; accumulator and a minutes count in the Y register.
; The subroutine uses the Miscellaneous Tools set, so the calling
; program must include _MTStartup and _MTShutDown.
;
;*****

        mcopy delay.macros

Delay   START

        phy                                ;Preserve the inputs during
        pha                                ; the ReadTimeHex call

        PushWord #0                        ;Read the time and date
        PushWord #0
        PushWord #0
        PushWord #0
        _ReadTimeHex

; Time and date values are on the stack. Remove them and store the
; second, minute, hour, and year values in memory. Discard the month,
; day, and day of the week.

        pla                                ;Get minutes and seconds
        tay                                ;Save this value temporarily
        and #$FF                            ; while I strip off the minutes
        sta OrigSecs                        ; and save the seconds
        tya                                ;Now retrieve the minutes value,
        xba                                ; put it in the low byte,
        and #$FF                            ; strip off the seconds,
        sta OrigMins                        ; and save the minutes

        pla                                ;Get years and hours
        tay                                ;Save this value temporarily
        and #$FF                            ; while I strip off the years
        sta OrigHrs                         ; and save the hours.
        tya                                ;Now retrieve the years value,
        xba                                ; put it in the low byte,
        and #$FF                            ; strip off the hours,
        sta OrigYrs                        ; and save the years

        pla                                ;Discard month and day,
        pla                                ; and day of the week

; Calculate the seconds value for the target time.

        pla                                ;Retrieve the seconds input
        clc                                ; and add the current secs. to it
        adc OrigSecs
        cmp #60                            ;Seconds < 60?
        blt SaveSecs
        sec                                ; No. Subtract 60
        sbc #60
        inc OrigMins                        ; and increment the minutes value
SaveSecs sta OrigSecs                      ;Save the final seconds value

```

266 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

; Calculate the minutes value for the target time.

```

        pla                ;Retrieve the minutes input
        clc                ; and add the current mins. to it
        adc OrigMins
        cmp #60            ;Minutes < 60?
        blt SaveMins
        sec                ; No. Subtract 60
        sbc #60
        inc OrigHrs        ; and increment the hours value
SaveMins sta OrigMins      ;Save the final minutes value

```

; Adjust the hours value, if necessary.

```

        lda OrigHrs
        cmp #24            ;Hours = 24?
        bne Repack
        stz OrigHrs        ; If so, set hours to zero

```

; Pack the final values into the byte-oriented format that
; ReadTimeHex produces.

```

Repack  lda OrigMins        ;Minutes and seconds
        xba
        ora OrigSecs
        sta TargetMinSec
        lda OrigYrs        ;Years and hours
        xba
        ora OrigHrs
        sta TargetYrHour

```

; Read the time until it is the same as or greater than the target.

```

NewTime PushWord #0        ;Read the time and date again.
        PushWord #0
        PushWord #0
        PushWord #0
        _ReadTimeHex

```

```

        pla                ;Put min/sec word in A
        ply                ; and year/hr word in Y
        plx                ;Ignore the other two words
        plx

```

```

        cpy TargetYrHour   ;Do year/hr words match?
        beq TryMS
        bcs ThatsAll
TryMS   cmp TargetMinSec    ; If so, compare min/sec words
        blt NewTime

```

```

ThatsAll rts              ;Exit

```

```

OrigSecs ds 2
OrigMins ds 2
OrigHrs  ds 2
OrigYrs  ds 2

```

```

TargetMinSec ds 2
TargetYrHour ds 2

```

END

Note that you need not assemble `DELAY.SRC` separately; the `Copy` statement makes the assembler append it to your program, then assemble it along with your own statements. However, you must still generate `DELAY`'s macro library (`DELAY.MACROS`) using `MACGEN`; an *mcopy* statement at the beginning of the subroutine reads the library when you assemble.

`DELAY` is just one example of code you might consider keeping separate, so you can access it from any program. In fact, I recommend putting *any* job you do frequently into a file of its own. That way, you build a program library of useful routines, and you needn't "reinvent the wheel" with each new program. Having working, debugged code in separate modules also makes your program listings shorter and (unless you have used *lots* of external code) easier to read.

Working with Color Tables

Within the Apple IIGS memory are 32 color tables: 16 tables for the 640 mode and another 16 for the 320 mode. So far, I have been working with table 0, the default, or "standard," color table for each mode, but you can easily make QuickDraw use any of the other 15.

Recall from Figure 7-5 that the low 4 bits of the *scan line control byte* (SCB) select the color table. Therefore, to start QuickDraw in 640 mode with, say, color table 5, you would push an SCB value of \$85 — rather than \$80 — onto the stack before calling `QDStartup`.

You can also switch color tables within a program by calling `SetMasterSCB` (see Table 7-15). For example, to switch to table 5 in 640 mode with no fill, enter:

```
PushWord #$85      ;640 mode, no fill, table 5
_SetMasterSCB
```

As I write this, only the first seven predefined color tables (tables 0 through 6) for each mode actually hold useful color data; the rest (tables 7 through 15) are filled with zeros. Tables 7-2 and 7-3 summarized the standard color table (0) for each mode; Table 7-16 summarizes the color data for tables 1 through 6 in 640 mode. (Only minipalettes 0 and 1 are shown; minipalettes 2 and 3 are identical to 0 and 1.) Note that in each case, only the Pixel 1 entries differ from the standard color table.

To actually *see* the dithered colors one of these alternate tables can produce, run my `DITHERS` program (Example 7-4) with the appropriate SCB

Table 7-15

__SetColorTable		Create a new color table
Call with:	PushWord <i>TableNumber</i> ;Number for new table (1-15) PushLong <i>TablePtr</i> ;Pointer to table __SetColorTable	
Result:	None	
Note:	TablePtr points to a 16-word table whose entries specify the color values for the pixel values 0 through 16.	
__InitColorTable		Get a copy of the standard color table
Call with:	PushLong <i>TablePtr</i> ;Pointer to table buffer __InitColorTable	
Result:	None	
Note:	TablePtr points to: ds 32 ;16-word buffer for table	
__GetColorTable		Copy one color table into another
Call with:	PushWord <i>TableNumber</i> ;Table to be copied (0-15) PushLong <i>DestTblPtr</i> ;Pointer to dest. table __GetColorTable	
Result:	None	
Note:	DestTblPtr points to: ds 32 ;16-word buffer for table	
__SetColorEntry		Set the value of a color in a color table
Call with:	PushWord <i>TableNumber</i> ;Table number (0-15) PushWord <i>EntryNumber</i> ;Pixel value (0-15) PushWord <i>NewColor</i> ;Master color value __SetColorEntry	
Result:	None	
__SetMasterSCB		Set the master scan line control byte
Call with:	PushWord <i>SCBValue</i> ;SCB value __SetMasterSCB	
Result:	None	
Note:	See the earlier Figure 7-5 for the layout of the SCB and note that bits 0 through 3 specify the color table.	

value in the ScreenMode equate. For example, to obtain the colors for table 5, set ScreenMode to \$85.

Color tables 1 through 6 for 320 mode contain the *same* color values as they do for 640 mode! (Again, this will probably change with time.) That is, the first eight entries in the 320 mode's table 1 produce black, light green, green, white, black, light green, yellow, and white, respectively — and so do the last eight entries in that table.

If none of the predefined color tables suit your needs, you can create your own. Since color values are 3 hex digits, or 12 bits long, they offer

Table 7-16

Pixel Value	Color	Master Color Value	Color	Master Color Value
<i>Table 1</i>		<i>Table 2</i>		
0	Black	000	Black	000
1	Light Green	0C0	Yellow	FF0
2	Green	0F0	Green	0F0
3	White	FFF	White	FFF
0	Black	000	Black	000
1	Light Green	0C0	Yellow	FF0
2	Yellow	FF0	Yellow	FF0
3	White	FFF	White	FFF
<i>Table 3</i>		<i>Table 4</i>		
0	Black	000	Black	000
1	Orange	F80	Red	F00
2	Green	0F0	Green	0F0
3	White	FFF	White	FFF
0	Black	000	Black	000
1	Orange	F80	Red	F00
2	Yellow	FF0	Yellow	FF0
3	White	FFF	White	FFF
<i>Table 5</i>		<i>Table 6</i>		
0	Black	000	Black	000
1	Rose	F0F	Bluish Black	004
2	Green	0F0	Green	0F0
3	White	FFF	White	FFF
0	Black	000	Black	000
1	Rose	F0F	Bluish Black	004
2	Yellow	FF0	Yellow	FF0
4	White	FFF	White	FFF

4,096 different number combinations. Hence, that's how many colors are available — 4,096.

You can create a new color table from scratch by issuing a `SetColorTable` call. `SetColorTable` requires you to supply the number of the new table and a pointer to a 16-word data structure containing the color values. Give your table any number from 1 to 15; don't number it 0, because that refers to the standard color table. Finally, give a `SetMasterSCB` to make QuickDraw use your new table.

If the standard color table is close to what you want, copy it into memory with an `InitColorTable` call, then assign the copy a table number with `SetColorTable`. Finally, use `SetColorEntry` calls to change entries in the copy to what you want, then `SetMasterSCB` to activate the table.

You can also use this procedure to create a new color table from an existing one that's similar. Simply give a `GetColorTable` call instead of `InitColorTable`.

CHAPTER 8

Events

In Apple IIGS terminology, an *event* is anything that requires the computer's attention. This is normally something the user does, such as pressing a key or the mouse button, but it could also be, say, a character coming in through the serial port. (No, it *doesn't* include a spider coming in through an expansion slot cutout on the back panel!)

The Event Manager keeps track of every event that occurs, and records information about it in an *event record* in memory. In most cases, the Event Manager also logs the type of event that occurred (e.g., key press or mouse button press) in an *event queue* in memory. The application program's job is, then, to check the event queue periodically and respond to any events that are meaningful to the program. This unique way in which the IIGS handles key presses, mouse clicks, and other events means that the structure of your programs must be somewhat different from programs you have written in the past.

Modal Programs

Most people write programs that are very mode-oriented. Consider, for example, a typical computer-assisted education program that drills a student in mathematics. The program starts by displaying a menu with the choices

Addition, Subtraction, Multiplication, Division, and Quit. When the student has selected an option, the program presents perhaps ten problems, one at a time, and the student enters the answer to each one. The student is then asked whether he or she wants to receive another ten problems or return to the menu.

Such a program is very inflexible. Once it starts presenting problems, the student has only two options: solve all ten of them or turn the computer off. And when the “Do you want more problems? Y/N” prompt appears, the student is forced to make another all-or-nothing decision — ten more problems or quit. Maybe he or she would like to continue, but wants only two or three problems. Unfortunately, the program doesn’t provide that choice.

Just as the student is always locked into one of several modes (choose from the menu, answer problems, or respond to a prompt), so is the program. It is entirely dedicated to displaying a menu, problem, or prompt, and processing the user’s response. While waiting for a response, the program does nothing; it just sits idle, locked in a mode.

Modal programs are probably so distracting because we humans don’t operate in modes; we can do several things at a time. For example, some people jot notes while they talk on the telephone or read a magazine while they soak in the bathtub. My teenagers even claim to be able to do their homework while watching TV! People don’t operate in modes, and neither should your programs.

In every program you write, then, you should allow the user to select any program function at any time. That is, the user should be able to pull down a menu, move things around on the screen, or do anything else when he or she *wants to*, without putting the program in a special user-interface mode. There should be no such modes. Instead, the program should regularly check, or *poll*, the event queue to find out whether the user has asked for the program’s attention.

For example, if the mouse pointer has been clicked in a box that closes a window, the Event Manager will put an entry in its event queue to reflect that action. In this case, upon polling the event queue and finding the mouse event, the program should remove the window from the screen immediately. Similarly, if the user has pressed a key in a program that displays text, the Event Manager will put a code for the key in the event queue. Upon finding it, your program should send that character to the screen.

In more general terms, then, the main part of your program should include statements that poll the event queue on a regular basis and act on any relevant event. To do this, you must design that main part as one large poll-and-respond loop. In Apple IIGS terminology, this is the *event loop*.

The Event Loop

The event loop must contain statements that do three things:

1. Read the event queue to find out whether an event has taken place.
2. Pull an event from the queue and determine its type (e.g., key press or mouse).
3. Perform the action the event signifies.

Of course, among the events that the program must check for is one indicating that the user wants to quit. This usually involves selecting “Quit” from a menu. When that happens, the program must exit the event loop, shut down the tools, and return to the calling program.

Until the user says to quit, however, the statements in the loop must be repeated indefinitely. Thus, while an “event-driven” program may seem to be simply waiting for something to happen, it’s actually racing through the event loop time and again, and polling the event queue during each pass.

Event Types

The Event Manager can keep track of seven types of events, plus a nonevent that indicates the queue is empty; in other words, no event has taken place. The Event Manager keeps information about each event, classifying it as to type.

Mouse Events

When someone presses the mouse button, the Event Manager posts it in the event queue as a *mouse-down event*. Similarly, releasing the mouse button gets posted as a *mouse-up event*. Generally, a mouse-up event is only significant for detecting the end of a dragging operation; for most simple applications, only mouse-down events are meaningful.

For both types of mouse events, the Event Manager also records the screen coordinates of the mouse pointer when the event occurred, because they may indicate something to your program. The Event Manager provides a `GetMouse` tool call that returns the mouse pointer coordinates.

Although the standard Apple IIGS mouse only has one button, some joysticks have two. For this reason, the Event Manager provides for two separate buttons, which it numbers 0 and 1 (where 0 is the button on the mouse).

Keyboard Events

When a user presses a letter, number, or symbol key on the main keyboard or the keypad, the Event Manager records it as a *key-down event*. Shift, Control, Caps Lock, Option, and Open-Apple (which are called “modifier” keys) are treated differently. Pressing them does not produce an event, but if you press one at the same time as a character key, the Event Manager reports that fact in the key’s event record. Knowing the state of the modifier key lets you distinguish uppercase and lowercase characters; for example, it lets you differentiate an *m* (only the M key was pressed) from an *M* (both Shift and M were pressed).

If you hold a character key down long enough to make it repeat, the Event Manager records that as an *auto-key event*. Thereafter, it records an additional auto-key event each time the key actually repeats. The Options selection in the Control Panel provides two parameters that let you change the keyboard’s repeat characteristics. Repeat Delay controls the amount of time it takes for a key to start repeating, while Repeat Speed controls the repetition rate.

Window Events

Window events are produced (as you may have guessed) by the Window Manager. There are two kinds of window events, activate and update.

An *activate event* results from a user activating or deactivating windows that overlap on the screen. Clicking the mouse pointer inside a partially covered window (to make it active) constitutes one activate event, and the currently active window becoming inactive constitutes a separate activate event. Thus, activate events generally occur in pairs.

Because an activate event signifies an action that must be performed as soon as possible — that is, activate or deactivate a window — the Event Manager doesn’t even post it in the event queue. Instead, the Event Manager constructs an event record for it and notes its occurrence at the top of a priority list (discussed shortly). When your program next asks for an event from the event queue, the Event Manager says, “Forget the event queue, my friend, I have something more important”, and hands the activate event to the program.

The other type of window event, called an *update event*, generally follows an activate event. With an update event, the Window Manager tells the application program (via the Control Manager) that one or more windows on the screen must be redrawn — usually because the user has opened, closed, activated, or moved a window. The Window Manager does the actual drawing, as you shall see later, but it takes its “get started” cue from the Control Manager.

As with activate events, the Event Manager creates an event record for each update event, but does not post its occurrence in the event queue. Update events are placed at the bottom of the Event Manager's priority list, however; even events in the queue are deemed more significant.

Switch Events

Just as the Window Manager generates window events, the Control Manager generates *switch events*. It produces a switch event when the user presses the mouse button (or sometimes a key) to indicate that he or she wants to switch from this application to another. In effect, a switch event tells the Control Manager to inform the program that it should do whatever housekeeping is necessary (e.g., update the screen and save critical information) before switching to the new application.

Like window events, switch events produce only an event record and are not entered in the event queue. However, since switching applications is such a drastic action, the Event Manager puts it second on the priority list, just below activate events.

User-Defined Events

The Apple IIGS ROM contains all the necessary firmware for communicating with standard peripheral devices, such as printers and modems. However, if some nonstandard device (say, an electronic piggy bank) is connected to your computer, you must write a *device driver* program to communicate with it. Input from such devices should be posted to the event queue as *device driver events*, using the PostEvent tool call.

The Event Manager can also accommodate up to four other general-purpose events that you can use. Your program should post them to the event queue as *application events*, using the PostEvent call.

Desk Accessory Events

A *desk accessory event* is generated when you press the OpenApple-Control-Esc key combination. Pressing these keys brings on the classic desk accessory menu, the one you use to obtain the Control Panel. You can ignore desk accessory events, because your program will never receive them; they are intercepted and handled by the Desk Manager.

The Null Event

Once your program retrieves an event, it should do whatever that event signifies (if anything), then go back and ask the Event Manager for the next

event. It should continue retrieving and processing events until the Event Manager returns a *null event*. This indicates that there are no more events to process, either in or out of the event queue. Indeed, null events are the best kind of all!

Event Priorities

As I mentioned in the preceding section, the Event Manager posts most events in the queue, but it places activate and update events from the Window Manager, and switch events from the Control Manager, in a priority list. When your program polls the Event Manager with a *GetNextEvent* call (described later), it returns the event record of the event that has the highest priority in its list. In order of decreasing importance, the priorities are:

1. Activate events (a window becoming active or inactive).
2. Switch events (the user wants to switch applications).
3. Mouse-down, mouse-up, key-down, auto-key, device driver, application-defined, and desk accessory events. The Event Manager posts all of these events in the event queue.
4. Update events, in front-to-back window order.
5. Null event.

Thus, when you tell the Event Manager to *GetNextEvent*, it begins by checking for a pending activate event. If one is available, it is passed to your application. Because of the way activate events are generated, there can never be more than two pending at the same time — one for a window being activated and the other for a different window being deactivated.

If no activate events are pending, the Event Manager looks for a pending switch event. Since a switch event signifies that the user wants to switch applications, the Event Manager temporarily forgets about priorities and digs down to the bottom of the list to see if any update event is pending. If an update is available, the Event Manager passes *it* (rather than the switch event) to your program. This signals the program to update all the windows before the switch takes place. That way, the windows will be intact if the user later returns to this application.

In the absence of pending activate and switch events, the Event Manager looks into the event queue. It always posts events in the queue in the order

it detects them. It also removes events *from* the queue in the same order, with the “oldest” event being removed first. Hence, the event queue operates like a vending machine; the package that was loaded into the machine (queue) first will be the one dispensed when you push the button (call `GetNextEvent`).

In technical terms, a queue is a data structure that operates in “first-in/first-out” fashion. Compare this to a data structure you encountered earlier, the 65816’s stack, which operates in “last-in/first-out” fashion.

When the event queue has been emptied, the Event Manager looks for a pending update event. It should find one if an activate event ever occurred — provided, of course, that the user has not requested an application switch in the meantime. (Switch events deal with pending update events automatically.) Finally, if no events are pending, the Event Manager passes a null event to the application.

Event Records

Whenever the Event Manager detects an event of any kind (queue or non-queue), it creates an *event record* that contains all the pertinent information about that particular event. It is this record that your program receives when it issues a `GetNextEvent` call. An event record contains five fields:

What	word	(event code)
Message	long integer	(event message)
When	long integer	(elapsed time since start-up)
Where	point	(mouse location)
Modifiers	integer	(modifier flags)

What — Event Codes

The *What* field holds a number from 0 to 15 that identifies the type of event that occurred (see Table 8-1). Your program can use this number to call the routine associated with that particular kind of event (or continue polling, if it’s the null event).

In practice, your event loop should use the event code to obtain the address of the appropriate event routine from a table, then execute that routine. Example 8-1 illustrates this approach.

This particular event loop accepts only mouse-down and key-down events, and calls `DoMouseDown` or `DoKeyDown` (not shown here) to

Table 8-1

0	- Null event
1	- Mouse-down
2	- Mouse-up
3	- Key-down
4	- Undefined
5	- Auto-key
6	- Update
7	- Undefined
8	- Activate
9	- Switch
10	- Desk accessory
11	- Device driver
12	- Application-defined
13	- Application-defined
14	- Application-defined
15	- Application-defined

Example 8-1

```

EventLoop  START
            using GlobalData

Again      anop                                ;Beginning of event loop
            lda QuitFlag                      ;Does user want to quit?
            bne AllDone                       ; If so, branch to AllDone
            ..                                (Otherwise, get next event record)
            ..
            lda EventWhat                     ;Get event code from event record,
            asl a                             ; double it,
            tax                               ; and put it in X

            jsr (EventTable,x)                ;Execute the event's routine
            bra Again

AllDone    rts

EventTable anop                                ;Event Manager events
            dc i'Ignore'                      ; 0 null
            dc i'JoMouseDown'                 ; 1 mouse-down
            dc i'Ignore'                      ; 2 mouse-up
            dc i'DoKeyDown'                   ; 3 key-down
            dc i'Ignore'                      ; 4 undefined
            ..                                (etc.)
            ..
            END

Ignore     START
            rts
            END

```


process them. All other events send the program to the Ignore subroutine, which has only one instruction, RTS. Note that because the addresses in EventTable are 2 bytes long, the event code must be doubled (by shifting it left one bit position) before it can be used as an index.

Message — Event Message

The *Message* field of an event record contains additional information about the event. The actual contents of the event message depends on what type of event it is, as follows:

Event Type	Event Message
Key-down	ASCII character code in low byte
Auto-key	ASCII character code in low byte
Mouse-down	Button number (0 or 1) in low byte
Mouse-up	Button number (0 or 1) in low byte
Activate	Pointer to window that generated event
Update	Pointer to window that needs redrawing
Device driver	User-defined
Application	User-defined
Switch	Undefined
Desk Accessory	Undefined
Null	Undefined

When — Elapsed Time

The event record's *When* field contains the elapsed time since you started the computer, in sixtieths of a second, or "ticks." This is handy for determining the order in which events occurred, in case you're interested.

Where — Mouse Pointer Location

The *Where* field gives the screen coordinates of the mouse pointer when the event occurred. These coordinates are meaningful for mouse-up events, because the Control Manager can use them to determine whether the pointer was in a significant area when the user released the mouse button.

Modifiers — Modifier Flags

The final item in an event record, *Modifiers*, is a word-size unit whose bits reflect the state of the modifier keys and certain other conditions when the event occurred. Figure 8-1 shows the arrangement of the bits in this *modifier*

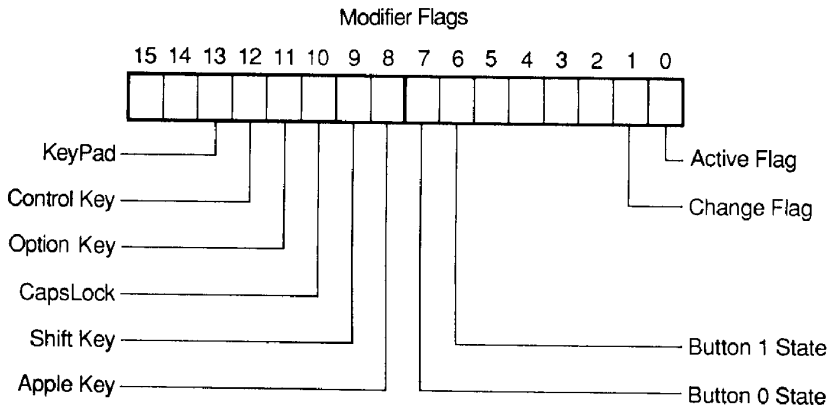


Figure 8-1

flags word. For each flag, a 1 indicates “true” (e.g., key is being pressed) and a 0 indicates “false” (e.g., key is unpressed).

Here, the upper byte (bits 8 through 15) reflects key events, while the lower byte (bits 0 through 7) reflects mouse and activate events. Control, Option, Caps Lock, Shift, and Apple are, of course, the modifier keys on the main keyboard. The Keypad flag (bit 13) is 1 if the user has pressed a key on the numeric keypad.

Button 0 State and Button 1 State track the two possible buttons on a controller; the button on the regular Apple IIGS is numbered 0. These bits behave the opposite of their keyboard counterparts; that is, a button bit set to 1 indicates that the button was in the *up* (unpressed) position, while a 0 indicates that it was *down* (pressed). Therefore, a mouse-down event will have the Button 0 State bit set to 0 and the Button 1 State bit set to 1.

Normally, the modifier flags are checked by an event routine; a subroutine that has been called from the program’s event loop. For example, suppose a key-down event has occurred, and you want to check whether the user has pressed OpenApple as well as the character key (perhaps to select from an on-screen menu). In response to a key-down event, the event loop calls, say, a DoKeyDown subroutine. To check for the OpenApple key, DoKeyDown would test whether bit 8 of the modifier flags is 1 (OpenApple down) or 0 (OpenApple up), and act accordingly. Thus, DoKeyDown may include this kind of instruction sequence:

```

        lda EventModifiers      ;Read modifier flags
        and #$100              ;Zero all but Open-
                                ; Apple key bit
        bne AppleDown          ;Is OpenApple key
                                ; pressed?
        . .
        . .
AppleDown . .                  ;Yes.
        . .

```

Event Manager Tool Calls

Table 8-2 summarizes the most common tool calls to the Event Manager.

Table 8-2

Housekeeping Calls	
<u>__EMStartup</u>	Start the Event Manager
Call with:	PushWord <i>DirectPageLoc</i> ;Loc. of 1-page work area
	PushWord <i>QueueSize</i> ;Max. number of queue events
	PushWord <i>XMinClamp</i> ;Leftmost column for mouse
	PushWord <i>XMaxClamp</i> ;Rightmost column for mouse
	PushWord <i>YMinClamp</i> ;Top row for mouse
	PushWord <i>YMaxClamp</i> ;Bottom row for mouse
	PushWord <i>ProgramID</i> ;The program's ID number
	<u>__EMStartup</u>
Result:	None
Note:	See text for descriptions of the inputs.
<u>__EMShutDown</u>	Shut down the Event Manager
Call with:	<u>__EMShutDown</u>
Result:	None
Calls That Access Events	
<u>__GetNextEvent</u>	Get the next event
Call with:	PushWord #0 ;Space for result
	PushWord <i>EventMask</i> ;Event mask
	PushLong <i>EventPtr</i> ;Pointer to event record
	<u>__GetNextEvent</u>
Result:	Word containing event code
Note:	See text for description of EventMask. EventPtr points to a buffer of the form:
	EventRecord anop

Table 8-2 (cont.)

Calls that Access Events (cont.)	
EventWhat	ds 2
EventMessage	ds 4
EventWhen	ds 4
EventWhere	ds 4
EventModifiers	ds 2

__FlushEvents	Remove specified events from the event queue
Call with: PushWord #0	;Space for result
PushWord <i>EventMask</i>	;Event mask
PushWord <i>StopMask</i>	;Stop mask
__FlushEvents	
Result:	Word value. If all events were removed, the word contains 0. If an event caused the removal operation to stop, the word contains its event code.
Note:	See text for descriptions of EventMask and StopMask.

Mouse-Reading Calls	
__GetMouse	Read the current mouse location
Call with: PushLong <i>MouseLocPtr</i>	;Pointer to a point buffer
__GetMouse	
Result:	None
Note:	MouseLocPtr points to a buffer of the form:
ds 2	;Row
ds 2	;Column

__Button	Test the up/down state of the mouse button
Call with: PushWord #0	;Space for result
PushWord <i>ButtonNum</i>	;Button number (0 or 1)
__Button	
Result:	1 if button is down, 0 if it is up.

Housekeeping Calls

The EMStartup call has some unusual-sounding inputs that are worth discussing. At the end of this section, I will tie everything together by listing the standard EMStartup sequence.

The Event Manager requires one page of working space in bank 0, and you obtain it (along with space for other tool sets) by issuing a NewHandle call to the Memory Manager. In the generalized program model (Example 6-2), the Event Manager's space follows the three pages that QuickDraw requires. Hence, that's the address to which *DirectPageLoc* should point.

The *QueueSize* input specifies the maximum number of events the queue can hold. A value of 0 sets up a default size of 20 events.

The *Clamp* inputs specify the area of the screen where the mouse position is meaningful. To make the mouse active throughout the entire screen, you would set *YMinClamp* to 0, *YMaxClamp* to 200, *XMinClamp* to 0, and *XMaxClamp* to the rightmost column — 320 for 320 mode or 640 for 640 mode.

Just as the program model employs a constant called *ScreenMode* to hold the master scan line control byte (SCB) value, it employs another constant, *MaxX*, to hold the *XMaxClamp* value. Constants are handy here, because they let you configure a new program for either 320 mode or 640 mode by changing only these two values in the model.

The final input, *ProgramID*, is the program identification number returned by the Memory Manager *MMStartup* call. The program model stores it in a variable called *MyID*.

The program model in example 6-2 uses the following sequence to start the Event Manager:

```

lda 4           ;Starting address of reserved space
clc            ;ZP to use = QD ZP + $300
adc # $300
pha           ;Push DirectPageLoc input
PushWord #20   ;Queue should accept 20 events
PushWord #0    ;Column clamp left
PushWord #MaxX ;Column clamp right
PushWord #0    ;Row clamp top
PushWord #200  ;Row clamp bottom
PushWord MyID  ;Program ID
_EMStartup
ldx #6
jsr PrepareToDie

```

Calls that Access Events

The *GetNextEvent* call copies the event record of the next available event into the buffer to which *EventPtr* points. The *EventMask* input is a word-size value that lets you specify the type(s) of events your program can process; Figure 8-2 shows its format. Here, a 1 in a bit position tells the Event Manager to pass events of that type to your application, while a 0 tells it to ignore those events. Thus, a program that's waiting only for a key press from the user would probably want to accept only key-downs. In this case, you would push an event mask value of 8 (binary 1000) onto the stack.

GetNextEvent returns a true/false indicator that contains a nonzero

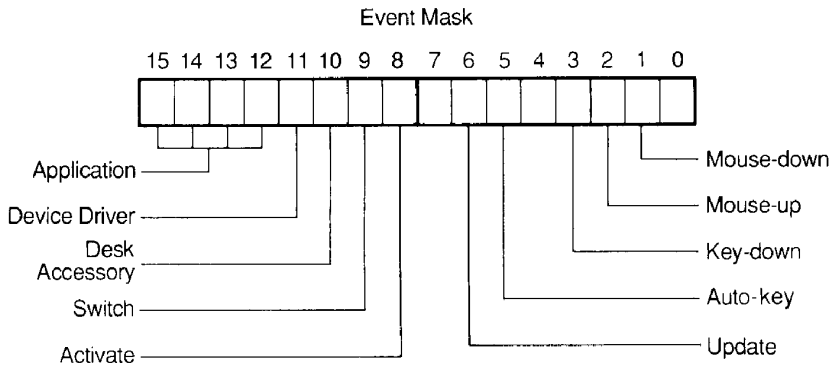


Figure 8-2

value if an event satisfies your event mask criteria or zero if no such events occurred. A nonzero value in the indicator signals your program to use the event code to call the routine that processes events of that type.

To illustrate, Example 8-2 shows a portion of an event loop for a program that accepts only key-down and auto-key events. Each key-down makes the program execute `DoKeyDown`; each auto-key makes it execute `DoAutoKey`.

`DoKeyDown` may do any of several things. If the key character (in the event record's Message field) represents text, `DoKeyDown` will simply display it on the screen. If the character indicates an option the user has selected from an on-screen menu, `DoKeyDown` will execute the routine that services that option — provided, of course, the character represents a valid menu choice.

It's important to realize that `GetNextEvent`'s event mask simply tells the Event Manager which types of events to pass to the application. Event types that are masked out (with a 0 in the event bit) still exist; the Event Manager still has records for them and queue-related events remain in the queue. This means that by using a mask, you run the risk of eventually filling up the queue. When the queue is full and the Event Manager detects a new event, it simply discards the earliest event in the queue — and that may well be an event your program needs!

There are several ways to guard against losing relevant events. The easiest way is to accept *every* event (using a mask of all 1's, or \$FFFF) and let your event table do the masking. In this case, the event table should contain a subroutine address for each event you want to accept and the address

Example 8-2

```

EventLoop  START
            using GlobalData
Again      anop                ;Beginning of event loop

            lda QuitFlag        ;Does user want to quit?
            bne AllDone         ; If so, branch to AllDone

            PushWord #0         ;Space for result
            PushWord ##28       ;Accept only key events
            PushLong #EventRecord ;Point to event record buffer
            _GetNextEvent

            pla                 ;Any key events?
            beq Again           ; No. Continue polling
            lda EventWhat       ; Yes. Read event code into A,
            asl a               ; double it,
            tax                 ; and put it in X

            jsr (EventTable,x)  ;Execute the event's routine,
            bra Again          ; then continue polling
AllDone    rts

EventTable anop                ;Event Manager events
            dc i'Ignore'        ; 0 null
            dc i'Ignore'        ; 1 mouse-down
            dc i'Ignore'        ; 2 mouse-up
            dc i'DoKeyDown'     ; 3 key-down
            dc i'Ignore'        ; 4 undefined
            dc i'DoAutoKey'     ; 5 auto key
            ..                  (etc.)
            ..
            END

Ignore     START
            rts
            END

DoKeyDown  START
            ..                  ;This routine is executed if
            ..                  ; a key-down event occurs
            rts
            END

DoAutoKey  START
            ..                  ;This routine is executed if
            ..                  ; an auto-key event occurs
            rts
            END

```

of a do-nothing “ignore” subroutine for all other entries, as in Example 8-2. This technique empties the queue automatically (provided you poll often enough to keep up with events), but it can be impractical for some applications.

Consider, for instance, an application in which only keystrokes are relevant in one part of the program, only mouse clicks are relevant in another part, and both keys and the mouse are acceptable in a third part. Here, masking with the event table would require the program to monitor the context in which each event occurred. You could instead make the event mask block out unwanted events and call *FlushEvents* to empty the queue when all required events have been received.

FlushEvents removes events from the queue based on two separate event masks. Specifically, it removes all events of the types marked 1 in *EventMask* up to (but excluding) the first event of any type marked 1 in *StopMask*. To clear the queue of all types specified by *EventMask*, use a *StopMask* value of 0. To clear the queue entirely, without regard to type, set *EventMask* to \$FFFF and *StopMask* to 0.

Mouse-Reading Calls

The two most useful mouse-related tool calls are *GetMouse* and *Button*. *GetMouse* reads the current mouse location into the record to which *MouseLocPtr* points. This location is given in the “local” coordinate system of the current *GrafPort*, which might be a window that occupies just a portion of the screen. This differs from the mouse location in the *Where* field of an event record, which gives the mouse’s “global” (screen) coordinates.

The *Button* call checks either of two buttons (0 or 1) on a controller and returns 1 if it is down (pressed) or 0 if it is up (unpressed). Recall that 0 is the button number for the standard Apple IIGS mouse.

A Simple Program that Uses the Event Manager

You will employ the resources of the Event Manager in virtually every program you write, so it’s time to look at a real program that uses it. Recall that the example programs in Chapter 7 use an Apple II-style routine that lets you exit a program by pressing any key. This is certainly a candidate for replacement by Event Manager calls.

However, instead of just adding a few minor embellishments to an old program, I will present a program that displays text as you type it, and quits when you press the Esc key. Admittedly, this won’t be a very elegant program, but it should tie this chapter together. So take a moment to examine Example 8-3, the listing of a program I call *SHOWTEXT*.

SHOWTEXT’s event loop calls *GetNextEvent* repeatedly until it receives a nonzero value from the stack. When that happens, *EventLoop*

Example 8-3

; SHOWTEXT displays text from the keyboard until the user presses Esc.

```

        absaddr on
        MCOPY showtext.macros

ShowText      START
              using GlobalData

;-----
;
; Global equates used throughout the program.

ScreenMode    gequ $80                ;640 mode, no fill
MaxX          gequ 640                ;640 mode for Event Manager

              phk                      ;Set data bank to program
              plb                      ; bank to allow absolute addressing

              jsr InitStuff            ;Initialize everything
              bcs AllDone              ;Quit if initialization fails

              stz QuitFlag             ;Initialize the quit flag to 0
              jsr EventLoop            ;Display the user's text

AllDone       anop                    ;All is done, shut down
              _DeskShutDown            ;Desk Manager
              _EMShutDown              ;Event Manager
              _QDShutDown              ;QuickDraw II
              _MTShutDown              ;Miscellaneous Tools

              PushWord MyID             ;Discard the program's handle
              _DisposeAll

              PushWord MyID             ;Memory Manager
              _MMShutdown              ;Tool Locator

              _Quit QuitParams          ;Do a ProDOS Quit call
              brk $F0                  ;If it fails, break

              END

;*****
;
; Global Data
;
;*****

GlobalData    DATA

QuitParams    dc i4'0'                ;Return to caller
              dc i'$4000'              ;Make program restartable in memory

ToolTable     dc i'NumTools'           ;No. of tool sets in table
              dc i'4,$0100'            ;QuickDraw
              dc i'5,$0100'            ;Desk Manager
              dc i'6,$0100'            ;Event Manager

TTEnd         anop

```

288 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

```

TableSize      equ TTEnd-ToolTable-2
NumTools       equ TableSize/4

EventRecord     anop                      ;Buffer for event record
EventWhat       ds 2
EventMessage    ds 4
EventWhen       ds 4
EventWhere      ds 4
EventModifiers  ds 2

MyID            ds 2                      ;This will hold the program's i.d.

VolNotFound     equ $45                  ;ProDOS error

QuitFlag        ds 2                      ;Quit flag
END

;*****
;
; InitStuff
;
; Gets space in bank zero for use as direct page by tools and
; then initializes QuickDraw and the Event Manager.
;
;*****

InitStuff       START
                using GlobalData

; Initialize the Tool Locator, Memory Manager, and Miscellaneous
; Tools.

                _TLStartup

                PushWord #0
                _MMStartup

                pla                      ;Memory Manager returns program's ID
                sta MyID

                _MTStartup              ;Misc. tools
                ldx #3
                jsr PrepareToDie

; Get some space for the direct page we need. QuickDraw needs
; three pages and the Event Manager needs one page.

                PushLong #0             ;Space for handle
                PushLong #$400          ;Four pages
                PushWord MyID           ;Owner
                PushWord #$C005         ;Locked, fixed, fixed bank
                PushLong #0             ;Location
                _NewHandle
                ldx #$FF
                jsr PrepareToDie

                pla                      ;Read handle and store in dp
                sta 0
                pla
                sta 2

```

```

        lda [0]                ;Get dp location from handle
        sta 4                  ; and store at loc 4 of dp

; Initialize QuickDraw

        pha                    ;Use dp obtained from handle
        PushWord #ScreenMode   ;Mode = 640
        PushWord #0            ;Use entire screen
        PushWord MyID
        _QDStartup
        ldx #4
        jsr PrepareToDie

; Initialize Event Manager

        lda 4                  ;dp to use = QD dp + $300
        clc
        adc #$300
        pha
        PushWord #20           ;Queue size
        PushWord #0            ;X clamp left
        PushWord #MaxX         ;X clamp right
        PushWord #0            ;Y clamp top
        PushWord #200          ;Y clamp bottom
        PushWord MyID
        _EMStartup
        ldx #6
        jsr PrepareToDie

; -----
;
; Load the RAM-based tools I need

LoadAgain    PushLong #ToolTable
              _LoadTools
              bcc ToolsLoaded

              cmp #VolNotFound
              beq DoMount
              sec
              ldx #$FE
              jsr PrepareToDie

DoMount      anop
              jsr MountBootDisk
              cmp #1
              beq LoadAgain

              sec
              rts

; The tools have been loaded. Initialize the Desk Manager.

ToolsLoaded  anop
              _DeskStartup

              clc                ;Clear the carry flag
              rts                ; and return

END

```

```

;*****
;
; Event Loop
;
; Display keyboard characters until user presses Esc.
;
;*****

EventLoop      START
                using GlobalData

                PushWord #20                ;Start the pen at (20,20)
                PushWord #20
                _MoveTo

Again          lda QuitFlag
                bne AllDone

                PushWord #0                ;Space for result
                PushWord #$FFFF            ;Accept any event
                PushLong #EventRecord      ;Point to event record buffer
                _GetNextEvent

                pla                        ;Is an event available?
                beq Again                  ; No. Continue polling
                lda EventWhat              ; Yes. Read event code into A,
                asl a                      ; double it,
                tax                        ; and copy it into X.

                jsr (EventTable,x)         ;Execute the event's routine,
                bra Again                  ; then resume polling

AllDone        rts

EventTable      anop                      ;Event Manager events
                dc i'Ignore'              ; 0 null
                dc i'Ignore'              ; 1 mouse down
                dc i'Ignore'              ; 2 mouse up
                dc i'DoKeyDown'            ; 3 key down
                dc i'Ignore'              ; 4 undefined
                dc i'DoAutoKey'            ; 5 auto-key
                dc i'Ignore'              ; 6 update
                dc i'Ignore'              ; 7 undefined
                dc i'Ignore'              ; 8 activate
                dc i'Ignore'              ; 9 switch
                dc i'Ignore'              ; 10 desk acc
                dc i'Ignore'              ; 11 device driver
                dc i'Ignore'              ; 12 ap
                dc i'Ignore'              ; 13 ap
                dc i'Ignore'              ; 14 ap
                dc i'Ignore'              ; 15 ap
                END

;*****
;
; Ignore
;
; This do-nothing subroutine returns to the event loop for
; events we don't want.
;
;*****

```

```
Ignore    START
          rts
          END
```

```
;*****
;
; DoKeyDown
;
; If user pressed Esc, DoKeyDown sets the QuitFlag to 1. It sends
; other keys to the screen, provided they are displayable.
;
;*****
```

```
DoKeyDown    START
              using GlobalData

              lda EventMessage      ;Read the ASCII character
              cmp #27               ;Is it Esc?
              bne NotEsc

              sta QuitFlag          ; Yes. Quit
              rts

NotEsc        anop                  ; No. See if it's displayable
              cmp #$20              ; (in the range $20 to $7E)
              bcc Exit
              cmp #$7F
              beq Exit

              pha
              _DrawChar              ;Display the character

Exit          rts
              END
```

```
;*****
;
; DoAutoKey
;
; Display this key on the screen.
;
;*****
```

```
DoAutoKey    START
              using GlobalData

              lda EventMessage      ;Read the ASCII character
              cmp #$20              ;To be displayed, a character
              bcc Leave             ; must be in the range $20
              cmp #$7F              ; to $7E
              beq Leave

              pha
              _DrawChar              ;Display the character
              rts

Leave          rts
              END
```

292 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

```
*****
*
* Prepare To Die
*
* Dies if carry is set.
*
* Error number to display is in X register.
* Assumes that Miscellaneous Tools is active.
*
*****

PrepareToDie  START
               bcs RealDeath      ;Carry = 1?
               rts                 ; No. Return to caller

RealDeath     phx                   ; Yes. Goodbye, program.
               PushLong #DeathMsg
               _SysFailMgr

DeathMsg      str 'Could not handle error '

               END

;*****
;
; Mount Boot Disk
;
; Tells the user to insert the disk that contains the RAM-based
; tools.
;
;*****

MountBootDisk START
               _Set_Prefix SetPrefixParams
               _Get_Prefix GetPrefixParams

               PushWord #0           ;Space for result
               PushWord #195         ;Column position for dialog box
               PushWord #30          ;Row position for dialog box
               PushLong #PromptStr   ;Prompt at top of dialog box
               PushLong #VolStr      ;Volume name string
               PushLong #OKStr       ;String in Button 1
               PushLong #CancelStr   ;String in Button 2
               _TLMountVolume

               pla                   ;Obtain the button number
               rts                   ; and return to caller

PromptStr     str 'Please insert the disk.'
VolStr        ds 16
OKStr         str 'OK'
CancelStr     str 'Shutdown'

GetPrefixParams dc i'7'
               dc i4'VolStr'
SetPrefixParams dc i'7'
               dc i4'BootStr'

BootStr       str '*/'

               END
```

uses the event code to obtain a subroutine address from an event table. Only two kinds of events are meaningful here: key-downs (which call `DoKeyDown`) and auto-keys (which call `DoAutoKey`).

`DoKeyDown` reads the key's ASCII character code from the event record's Message field and sets a quit flag if the code is 27, or Esc. (See Appendix B for the ASCII character codes.) If the key is something other than Esc, `DoKeyDown` tests whether the code is displayable (i.e., whether it's between \$20 and \$7E) and, if so, displays it with a QuickDraw `DrawChar` call.

`DoAutoKey` is identical to `DoKeyDown`, except it does not check for Esc.

Again, `SHOWTEXT` is not very elegant, and you could probably improve it. For example, it doesn't detect the end of a line; it simply keeps sending characters to the screen, without accounting for the fact that you can't see them. To eliminate this problem, you could follow each character-display operation with a QuickDraw `GetPen` call, and move the pen to the next line if the column position is equal to or greater than the rightmost column of the screen (column 319 or 639).

You could also modify the program to accept the Tab and Return keys, where Tab moves the pen to the next predefined tab stop and Return moves it to the beginning of the next line. In fact, by adding enough enhancements, you could actually expand `SHOWTEXT` into a full-fledged word processor! As textbooks often say, "This exercise is left to the reader."

What's Next?

The programs in this chapter and the preceding one illustrate the techniques you should use to communicate with Apple IIGS tool sets and show some rudimentary ways to display graphics and process events. However, none of these programs provide meaningful interaction with the user. Except for `SHOWTEXT`, the user can only view what the program puts on the screen, and press a key when he or she wants to quit. Surely, you will want to be able to do more than that!

For example, you may want to show portions of several pictures on the screen, in overlapping fashion, and let the user choose the one he or she wants to see. You may also want your program to present a menu upon demand, to let the user indicate with the mouse what should be done next. To do these kinds of tasks, you need the services of tool sets I have not yet described. Let's take them one at a time, starting with the Window Manager.

CHAPTER 9

Working with Windows

The programs presented thus far work with only a single object — a graphics picture or a block of text — on the screen. However, sometimes you may want to let the user select from several different objects. To deal with multiple objects, either pictures or text, you must put them in individual “windows” — and for that you need the services of the *Window Manager*.

A window is an object on the screen that contains graphics or text, or both. Usually a window shows an entire data area (e.g., an entire picture), but if the data area is too large to fit on the screen, the window can show just a portion of it. Thus, a window can represent the user’s view of a larger object, just as a window in your home or office is your view of a *much* larger object: the world outside!

The Apple IIGS provides two types of standard windows, called *document* and *alert* (see Figure 9-1). Alert windows are produced by the Dialog Manager; they are described in Chapter 12. The Window Manager displays document windows, so that’s the kind that interests us at the moment. To save you from having to read “document windows” throughout the rest of this chapter, I will simply refer to them as “windows.”

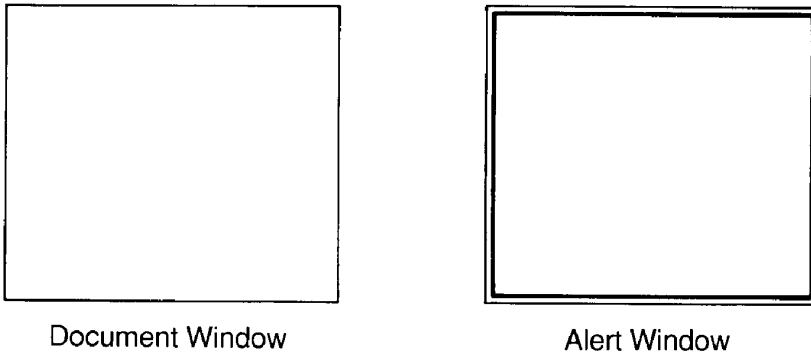


Figure 9-1

Window Components

You can make a window as plain or elaborate as you want, and the Window Manager offers a variety of standard components you can add to windows. Some components are *controls* that let the user operate on the information being displayed, or on the window itself, by clicking the mouse button. Other components are informational. Figure 9-2 shows a window that's equipped with all the predefined (or standard) components.

Content Region

Every window has a content region. It is the window's main area, the place where it displays your graphics or text. That will be the entire window if you haven't added other components.

Title Bar

The title bar displays the window's name or title. It is also a control region; the user can move the window — perhaps to reveal a window beneath it — by simply dragging it by its title bar.

Close and Zoom Boxes

The title bar can include a close box at the left or a zoom box at the right, or both. Clicking the mouse in a *close box* makes the window retreat into an icon or simply disappear. It's up to your application program to control what

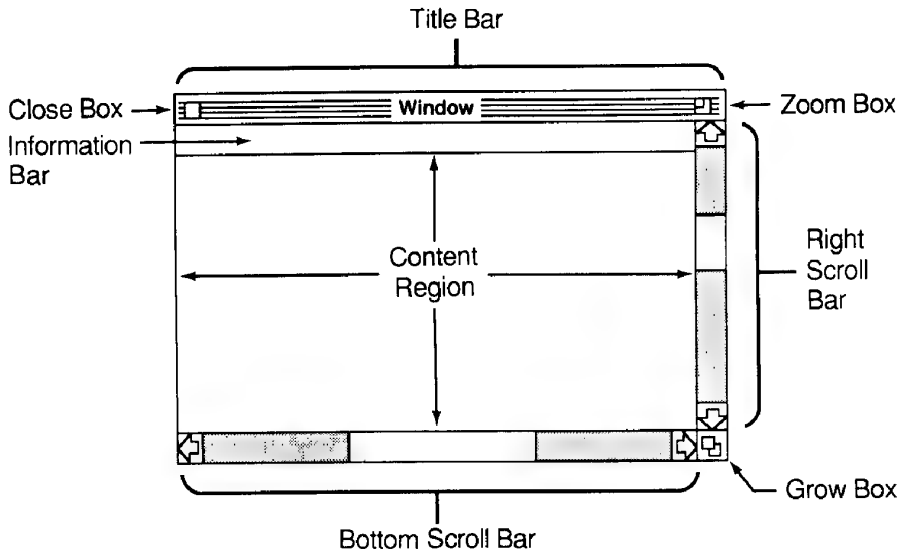


Figure 9-2

happens when a window is closed, but in general, you should simply hide the window (make it invisible), so it can be reopened later.

The application program determines the initial size of a window and its position on the screen. As just mentioned, however, you can change a window's position by dragging it by the title bar. Another control, the *zoom box*, lets you change a window's size. (A grow box also lets you resize a window, but is a more versatile control than a zoom box — more about that shortly.)

By clicking in the zoom box, the user can expand a window to some maximum size that's been preset by the program. (Most application programs make windows zoom to fill the entire screen.) Once a window has been zoomed to its maximum size, clicking in the zoom box again shrinks the window back to its previous size and position.

Grow Box

While the zoom box is like an on/off switch that produces only two window sizes (original size or full size), the grow box lets you expand or shrink a window incrementally, to whatever size you want. To resize a window, put the mouse pointer in its grow box and drag the box to where you want it to end up. Moving away from the window makes it expand and show more of

the pixel image; moving toward the window, or into it, makes it contract and show less of the image.

Once you have resized a window using the grow box, it retains that size until you drag the grow box again. Thus, while zooming a “grown” window still expands it to full size, zooming a full-size window shrinks it to its *grown* size — not to its initial size.

Scroll Bars

If your picture is too large to fit on the screen, you can display just a portion of it in a window and provide scroll bars to let the user work his or her way through the rest of the data area. A scroll bar has arrows at each end and a white rectangular “scroll box” in the gray area between the arrows.

Clicking in one of the right scroll bar’s arrows “moves” the data area upward or downward beneath the window. Similarly, clicking in the bottom scroll bar’s arrows moves it to the right or left. (In each case, the number of rows or columns the data area actually moves must be set by the application program.)

Thus, scroll bar arrows function like the control knobs on a microfiche viewer. Just as you rotate the “vertical” knob on a viewer to make a higher or lower portion of the fiche appear on the screen, you click the right scroll bar’s top or bottom arrow to move the data area upward or downward beneath the window’s content region. I could also compare a viewer’s “horizontal” knob to the bottom scroll bar’s left and right arrows, but you get the idea.

The *scroll box* or *thumb* within each scroll bar indicates the current position of the content region within the larger data area. That is, the scroll boxes show how far down or to the right the content region is from the beginning of the data area (see Figure 9-3). As the user “moves” through the data area by clicking a scroll arrow, the scroll box moves vertically or horizontally to reflect the change in position.

The size of a scroll box is also meaningful insofar as it indicates how high or wide the data area actually is. If the data area is very high (i.e., long), the right scroll box will be small; it will cover only a small portion of the gray area. Conversely, if most of the data area is already on the screen, the right scroll box will cover most of the gray area. Thus, the scroll boxes not only indicate the relative position of the content region, but also how much of the data area is being displayed.

Besides being position and size indicators, the scroll boxes are also movement controls; they do the same thing as the scroll bar arrows, but

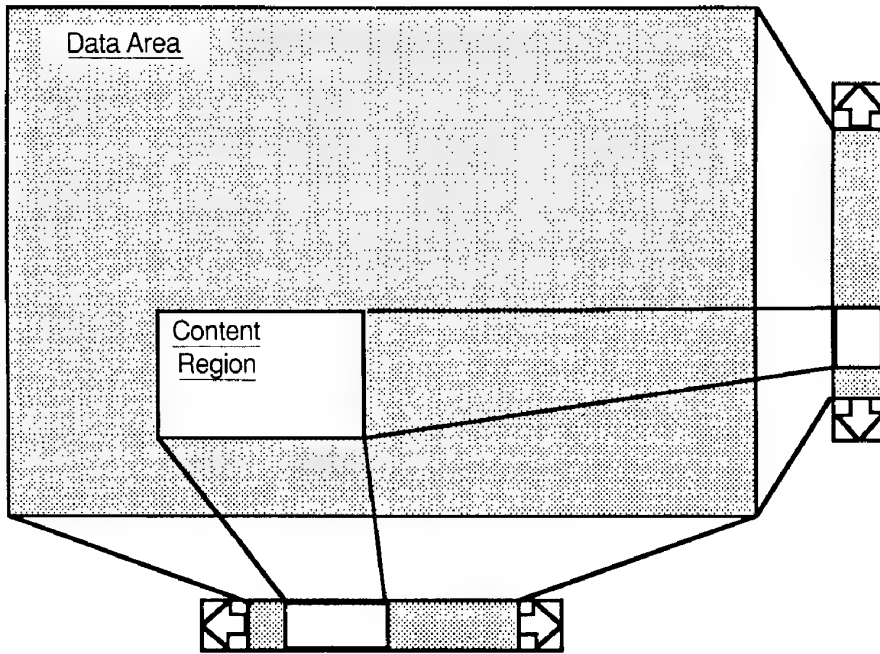


Figure 9-3

faster. To move vertically or horizontally through a data area, simply drag the scroll box up or down, left or right, to where you want it. The instant you release the mouse button, the content region will change to show the portion of the data area you have selected.

The gray area in a scroll bar is also a movement control; clicking in it moves the data area by a “page” worth of pixels. By convention, this means the data area moves ten times as far as it does when you click in a scroll bar arrow. Thus, if clicking the up arrow in the right scroll bar displays the next four rows of data, clicking in the top gray area displays the next 40 rows.

In short, there are three ways to move a data area under a window: by clicking in an arrow to scroll incrementally, by clicking in the gray area to “page” a window at a time, and by dragging the scroll bar a selected distance. Figure 9-4 illustrates these three techniques.

Information Bar

The information bar is an optional strip of space below the title bar that your program can use to display additional graphics or text. You could use it to

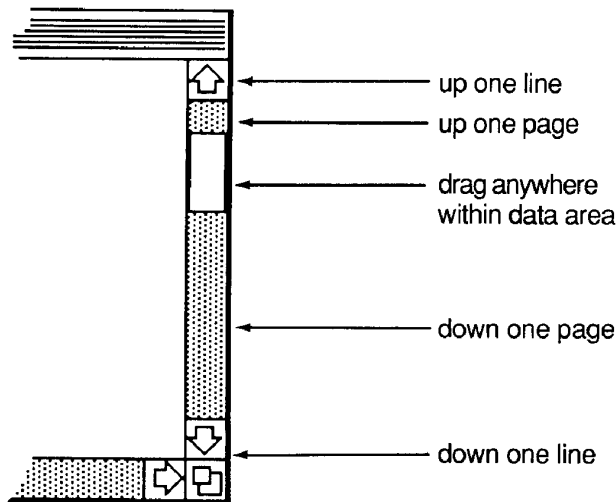


Figure 9-4

display something simple, such as messages to the user, or something more complex, such as the “scoreboard” in a computerized football game, a treasure map in an adventure game, or a bar chart in a business program.

In short, the information bar is handy for displaying anything you want to keep separate from the content region. Of course, you pay a price for this convenience in that the information bar takes up space that would otherwise be part of the content region.

Active and Inactive Windows

Although you can display several windows on the screen at the same time, only one of them may be *active*; the rest are *inactive*. The active window is the only one the user can operate on — that is, move, scroll, draw in, or whatever. To activate an inactive window, simply click the mouse anywhere in it. This moves the window to the front and displays any controls it has. As you may recall from Chapter 8, it also generates an activate event, the event that has the highest priority.

Fundamental Tool Calls for Windows

To display a window, your program must do these things:

1. Start QuickDraw and the Event Manager — the Window Manager needs the services of both of them.
2. Start the Window Manager with a WindStartup call.
3. Define the window with a NewWindow call. NewWindow creates a GrafPort for the window and stores the entire GrafPort record plus information about the window's characteristics (e.g., its size and position and which controls it has) in a large *window record* in memory.
4. Activate the window's GrafPort with a SetPort call to QuickDraw.
5. Draw the image the window is to display, using Quickdraw calls.
6. Give a ShowWindow call to display the completed window.

Table 9-1 summarizes these Memory Manager tool calls, along with some others that application programs need to work with windows. The *Housekeeping Calls* include WindStartup and WindShutDown, plus calls that let you create and display a new window and close an existing one. Of these six tool calls, NewWindow is the most complex.

NewWindow

The NewWindow call requires you to supply a pointer to a parameter table for the window you want to create, and returns a pointer to the GrafPort it sets up for that window. Table 9-2 lists the entries in NewWindow's parameter table. The table is quite long (24 entries in all), but don't be intimidated by it. Unless you're creating a window that uses all the controls, many of the entries will be zero. Let's take the entries in the order they appear in the table.

The first parameter, param__length, gives the total length of the table (including this entry) in bytes. NewWindow checks param__length against the byte count it expects and reports an error if they don't match. This ensures that you have supplied the correct number of entries. The easiest way to get the paramlength value is to label the beginning and end of the table, and let the assembler calculate the length, as in this example:

```

WinlParamTable anop
    dc i'WinlEnd-WinlParamTable' ;Length of table
    . .
    . . (23 more parameter table entries)
    . .
WinlEnd anop

```

Table 9-1

Housekeeping Calls

__WindStartup	Start the Window Manager
Call with: PushWord <i>ProgramID</i>	;The program's ID number
__WindStartup	
Result: None	
Note:	The Window Manager shares the Event Manager's space in bank 0.
__Refresh	Redraw the screen
Call with: PushLong #0	
__Refresh	
Result: None	
Note:	Clears the screen and redraws it. Generally, you should call Refresh immediately after you do a WindStartup.
__NewWindow	Create a new window using specified parameters
Call with: PushLong #0	;Space for result
PushLong <i>ParamPtr</i>	;Pointer to parameter table
__NewWindow	
Result: Pointer to new window's GrafPort (long word)	
Note:	See text for description of the window parameter table.
__ShowWindow	Make the specified window visible
Call with: PushLong <i>WindPortPtr</i>	;Pointer to window's port
__ShowWindow	
Result: None	
Note:	Makes the window visible and draws it. Does not change the front-to-back ordering of the windows; see also SelectWindow.
__CloseWindow	Close the specified window
Call with: PushLong <i>WindPortPtr</i>	;Pointer to window's port
__CloseWindow	
Result: None	
Note:	Removes the window from the screen and destroys its window record.
__WindShutDown	Shut down the Window Manager
Call with: __WindShutDown	
Result: None	

Table 9-1 (cont.)

Window-Shuffling Calls	
<code>__HideWindow</code>	Make the specified window invisible
Call with: <code>PushLong WindPortPtr</code> ;Pointer to window's port	
<code>__HideWindow</code>	
Result: None	
Note: If this is the frontmost (active) window, <code>HideWindow</code> makes the window behind it active and generates the appropriate activate events.	
<code>__SelectWindow</code>	Make the specified window active
Call with: <code>PushLong WindPortPtr</code> ;Pointer to window's port	
<code>__SelectWindow</code>	
Result: None	
Note: Unhighlights the currently active window, brings the specified window to the front, highlights it, and generates the appropriate activate events. Should be called if there's a mouse-down event in the content region of an inactive window.	

The *wFrame* parameter is a word-size bit pattern that is arranged as shown in Figure 9-5. Here, Title Bar, Close Box, Right Scroll Bar, and so on, are switches by which you specify the controls the window should have; a 1 means the window has that control, a 0 means it does not. A 1 in the "Movable" bit specifies that the window may be dragged by its title bar. The "Visible" bit stipulates whether the window should be visible (1) or invisible (0). Generally, you should make it invisible to start. Then, after you have drawn its contents, do a `ShowWindow` call to make it visible.

Some of these bits are interrelated, and you should observe the following rules when constructing a *wFrame* word:

- Since the close box and zoom box are part of the title bar, if the window is to have either or both boxes, it must also have a title bar. In other words, if bit 8 or 14 is set to 1, bit 15 must also be set to 1.
- A window must also have a title bar if it is to be movable (bit 7 = 1).
- To have a grow box, a window must also have a scroll bar. Thus, to set bit 10 to 1, bit 11 or 12 must also be 1.

Because each bit is significant, you should set up *wFrame* with a DC directive whose operand is a 16-bit binary (%) pattern. Some examples are:


```

wFrame dc i '%0000000000000000'      ;No controls
wFrame dc i '%1000000010000000'      ;Title bar
                                         ; (can be dragged)
wFrame dc i '%1100000110000000'      ;Title bar, close
                                         ; and zoom boxes
wFrame dc i '%1101110110100000'      ;All controls and
                                         ; information bar

```

Using a binary pattern not only helps make your parameter table more understandable, but it makes troubleshooting easier if a particular control (say, the zoom box) is missing when the window is displayed.

The next item in the parameter table, *wTitle*, is a pointer to the text for the window's title (or 0, if the window has no title bar). Set up *wTitle* with this kind of directive:

Table 9-2

Name	Data Type	Description
param__length	Word	Number of bytes in parameter table.
wFrame	Word	Bit vector that describes the window.
wTitle	Long word	Pointer to window's title.
wRefCon	Long word	Reserved for application's use.
wZoom	Rect	Position of content when zoomed.
wColor	Long word	Pointer to window's color table.
wYOrigin	Word	Vertical offset of content region.
wXOrigin	Word	Horizontal offset of content region.
wDataH	Word	Height of data area (rows).
wDataW	Word	Width of data area (columns).
wMaxH	Word	Maximum height to which content region can grow.
wMaxW	Word	Maximum width to which content region can grow.
wScrollVer	Word	Number of pixels to scroll content vertically.
wScrollHor	Word	Number of pixels to scroll content horizontally.
wPageVer	Word	Number of pixels to page content vertically.
wPageHor	Word	Number of pixels to page content horizontally.
wInfoRefCon	Long word	Value passed to information bar draw routine.
wInfoHeight	Word	Height of information bar (rows).
wFrameDefProc	Long word	Pointer to routine that draws the window's shape.
wInfoDefProc	Long word	Pointer to routine that draws the information bar's interior.
wContDefProc	Long word	Pointer to routine that draws the content region's interior.
wPosition	Rect	Content region's screen coordinates.
wPlane	Long word	Window's order.
wStorage	Long word	Address of memory to use for window record.

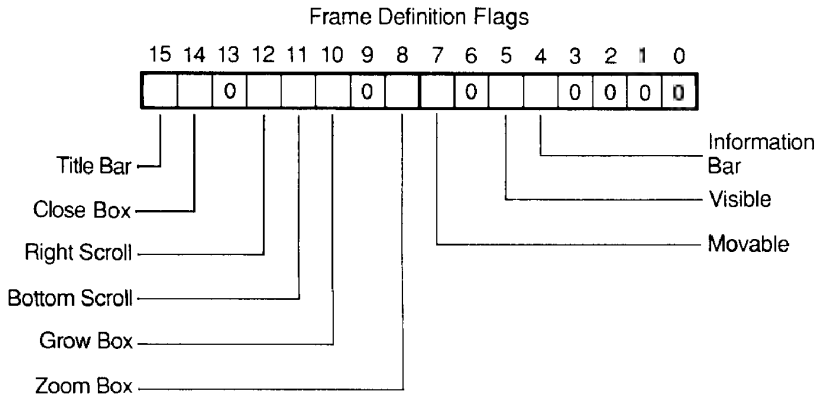


Figure 9-5

```
wTitle dc i4'WinlTitle'
```

where `WinlTitle` points to a statement such as:

```
WinlTitle str 'Window 1'
```

`wRefCon` is an application-defined reference value. In general, you should enter `dc i4'0'`.

`wZoom` defines the screen coordinates of the content region when the window is zoomed. This takes a directive of the form:

```
dc i'V1,H1,V2,H2'
```

Generally, you will want the zoomed window to fill the entire screen, so if it has no title bar or scroll bars, you would enter (in 640 mode):

```
dc i'0,0,199,639'
```

If your window has a title bar or scroll bar, however, you must account for the space they occupy. The title bar is 26 rows high (rows 0 through 25), the bottom scroll bar is nine rows high (191 through 199), and the right scroll bar is 19 columns wide in 640 mode (621 through 639). Hence, the `wZoom` rectangle for a 640 mode window that has only a title bar is defined by:

```
dc i'26,0,199,639'
```

A window with a title bar and both scroll bars is defined by:

```
dc i '26,0,190,620'
```

wColor points to the color table used to draw the window's frame. Entering 0 calls up the default color table.

wYOrigin and *wXOrigin* are the number of rows and columns the content region is offset from the top left-hand corner of the data area. If you set both values to 0 (the usual case), the window's content region will show the beginning of the data area.

wDataH and *wDataW* are the height and width of the data area; that is, the entire picture.

wMaxH and *wMaxW* are the maximum height and width to which the window can grow. To let the window fill the screen, enter 200 for *wMaxH* and either 320 or 640 for *wMaxW*. If the window has no grow box, enter 0 for both parameters.

wScrollVer is the number of pixels the content region should scroll vertically when the user clicks in one of the right scroll bar's arrows. Make this a small number, such as 4. Similarly, *wScrollHor* is the number of pixels the content region should scroll horizontally when the user clicks in one of the bottom scroll bar's arrows. Because the screen is wider than it is tall, make *wScrollHor* larger than *wScrollVer*; a value of 8 would be appropriate. Of course, if your window has no scroll bars, set both parameters to 0.

wPageVer and *wPageHor* are the number of pixels the content region should scroll (or "page") when the user clicks in a scroll bar's gray area. These values should be much larger than *wScrollVer* and *wScrollHor*. It is appropriate to make them, say, ten times larger, in which case *wPageVer* and *wPageHor* would be 40 and 80, respectively. As before, if your window has no scroll bars, set both parameters to 0.

wInfoRefCon is a long-word value passed to the routine that fills in the information bar. It may be, perhaps, a pointer to a string the routine should display. *wInfoHeight* is the height of the information bar. Make both parameters 0 if your window has no information bar.

The next three parameters hold pointers to various drawing routines. *wFrameDefProc* points to a routine that defines the window's shape. You would only use it if you want the window to have a shape other than the standard rectangle; in general, set *wFrameDefProc* to 0. *wInfoDefProc* points to a routine that fills in the information bar.

wContDefProc points to a routine that draws the window's content region. The routine must end with an RTL (Return from Subroutine Long)

instruction because it is called by the Window Manager, which may reside in a different bank than your program.

wPosition holds the initial screen coordinates of the window's content region, which specify its position and size. These coordinates become the PortRect of the window's GrafPort.

wPlane indicates where this window is to appear in a stack of overlapping windows. Usually you should create the windows in back-to-front order, and set *wPlane* to -1 in each window's parameter table. But you needn't create all the windows at the same time. You can create a new window at any time by issuing a NewWindow call. If the new window is to appear atop the others, set its *wPlane* value to -1 ; if it is to appear beneath the others (at the bottom of the stack), set *wPlane* to zero.

The final parameter in the table, *wStorage*, points to the memory location where the window's record should be stored. You can allocate this space yourself, but because Apple may change the length of the window record sometime in the future, it's safer to let the Window Manager allocate the space. To hand this job over to the Window Manager, set *wStorage* to zero.

The window record that NewWindow creates is 325 bytes long, the longest record in an Apple IIGS program. Fortunately, you never work with it directly; rather, you access it indirectly, through calls to the Window Manager.

You also access the window record indirectly when you make QuickDraw tool calls, because the window's GrafPort record is part of its window record. In fact, NewWindow returns a pointer to this GrafPort, or *Window Manager port*, on the stack. As Table 9-1 shows, this pointer is required input for SelectWindow, HideWindow, and other window calls.

In short, then, NewWindow creates a window record in memory that contains a copy of the current GrafPort record, makes this window (and its Window Manager port) active, and returns a pointer to its port. Because the Window Manager port is active, any calls you make to QuickDraw will only apply to this port.

Having an individual port for each window is convenient in that any changes you make with QuickDraw (such as moving the pen or increasing its size) affect only the active window. Every other window retains its own drawing environment. Thus, you never have to worry about the current pen attributes, color table, or anything else when you switch windows; the window you activate will operate under its own port settings.

ShowWindow

The ShowWindow call displays the window NewWindow has created. Your program should do a ShowWindow after it has drawn a window's content region.

CloseWindow

The `CloseWindow` call deletes a window entirely by removing it from the screen and destroying its window record. If you simply want to hide a window, and keep it available for displaying later, you can make it invisible by calling `HideWindow`.

Window-Shuffling Calls

The `HideWindow` call makes a window invisible, but (unlike `CloseWindow`) keeps its record in memory. If the window is at the top of a stack of windows, `HideWindow` also makes the window beneath it active. Generally, your program should call `HideWindow` if the user clicks the mouse in the active window's close box.

The `SelectWindow` call makes a window active by bringing it to the front and highlighting it. You would normally do a `SelectWindow` if the user has pressed the mouse button in the content region of an inactive window. `SelectWindow` is also useful for revealing a window that was hidden previously. Here, the program should allow the user to unhide a window by selecting its name from a menu.

At this point, you may be expecting me to present a program that puts some overlapping windows on the screen and lets you shuffle them around. I have already described the tool calls necessary to do that, so it's reasonable to anticipate such an example. However, I will forgo the programming for the moment because it would entail using the Event Manager's `GetNextEvent` call. `GetNextEvent` would require you to do a lot of tasks manually that are performed automatically by a Window Manager resource called the `TaskMaster`.

The TaskMaster

The Window Manager's `TaskMaster` tool is an enhanced version of the Event Manager's `GetNextEvent` tool. In fact, `TaskMaster` actually calls `GetNextEvent` internally. Therefore, if the Window Manager is active (as it will be for most applications), you can *forget about* `GetNextEvent` entirely and use `TaskMaster` instead!

Like `GetNextEvent`, `TaskMaster` checks the Event Manager's event list and returns a 0 (the null event) if no event is pending. If an event *is* pending, however, `TaskMaster` does not simply pass it to your program, as `GetNextEvent` does, but tries to handle the event itself. For example, if the user

presses the mouse button in the active window's zoom box, TaskMaster does everything necessary to zoom the window to its predefined full size. Once it has zoomed the window, TaskMaster returns a null event to your program, as if the zoom operation had never occurred!

Calling TaskMaster

Here is a summary of the TaskMaster tool call:

```

_TaskMaster          Get the next event using TaskMaster
  Call with:  PushWord #0                ;Space for result
              PushWord EventMask       ;Event mask for
              ; GetNextEvent
              PushLong TaskRecPtr      ;Pointer to extended
              ; event record
              _TaskMaster
  Result:  Task code (word)

```

Note that like the Event Manager's GetNextEvent call, the Window Manager's TaskMaster call requires two inputs: an event mask and a pointer to an event record. The event mask is the same as the one used by GetNextEvent (see Figure 8-2), but TaskMaster's *task record* has two more entries than GetNextEvent's event record. Specifically, a task record contains these seven entries:

What	word	Event record entries, same as
Message	long word	for GetNextEvent
When	long word	
Where	point	
Modifiers	word	
TaskData	long word	Additional entries for TaskMaster
TaskMask	long word	

Here, TaskData is a long word in which TaskMaster returns information pertinent to a window event. This is usually a pointer to the window's port. In your programs, define TaskData with a DS 4 directive.

The TaskMask (shown in Figure 9-6) is a 32-bit pattern whose 13 low bits indicate which operations you want *TaskMaster*, rather than your program, to perform. You can, for example, tell TaskMaster to do everything necessary when the user scrolls, drags, closes, zooms, or grows a window. You can also tell TaskMaster to redraw the active window and make it inactive

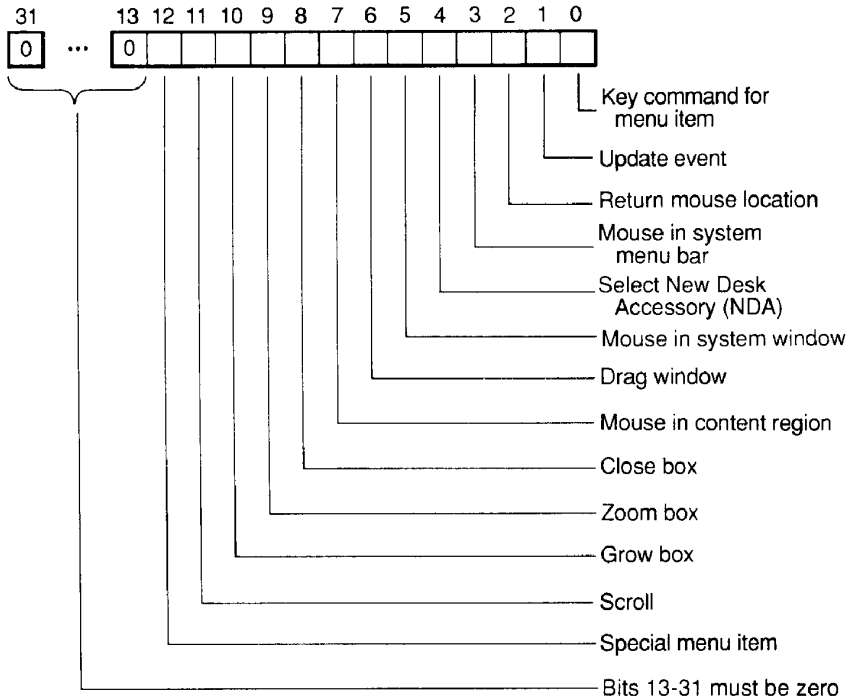


Figure 9-6

(i.e., *update* it) when the user clicks in an inactive window. To hand these tasks to TaskMaster, set all 13 low bits in the TaskMask to 1 by defining it with a DC `i4'$01FFF'` directive.

Although TaskMaster relieves your program of a lot of standard operations, the program must still deal with events that TaskMaster doesn't know how to handle. These are key-down, auto-key, activate, switch, desk accessory, and the user-defined events. If your program is to accept any of these events, it must include subroutines that process them.

Processing Events

TaskMaster returns a word-size *task code* (a number from 0 to 28) on the stack. If no event is pending, or if TaskMaster has already processed an event (as directed by your TaskMask), the task code is 0 — the number of our old friend, the null event. Otherwise, TaskMaster returns one of the codes listed in Table 9-3.

Table 9-3

Event Manager events

- 0 - Null event
- 1 - Mouse-down
- 2 - Mouse-up
- 3 - Key-down
- 4 - Undefined
- 5 - Auto-key
- 6 - Update
- 7 - Undefined
- 8 - Activate
- 9 - Switch
- 10 - Desk accessory
- 11 - Device driver
- 12 - Application-defined
- 13 - Application-defined
- 14 - Application-defined
- 15 - Application-defined

TaskMaster events

- | | | | |
|------|----------------------|----|---|
| 16 - | Mouse button pressed | in | desktop (screen) |
| 17 - | " | " | " |
| 18 - | " | " | system menu bar |
| 19 - | " | " | system window (e.g., desk accessory) |
| 20 - | " | " | content region of window |
| 21 - | " | " | drag region (title bar) |
| 22 - | " | " | grow box |
| 23 - | " | " | close box |
| 24 - | " | " | zoom box |
| 25 - | " | " | information bar |
| 26 - | " | " | right scroll bar |
| 27 - | " | " | bottom scroll bar |
| 28 - | " | " | frame, but in none of the above regions |
| | | | drop region |

The first 16 codes are the event codes that can result from TaskMaster making a `GetNextEvent` call to the Event Manager. Of these events, TaskMaster can process only `Update` (code 6) on its own. It passes all other codes to your application program, because, as after all, it has no way of knowing what you want these events to do.

The list of events your program must handle is normally quite short, however. For starters, you will probably never receive a mouse-down or mouse-up code, because when TaskMaster receives one, it says, "My boss, the Window Manager, will want to know about this," and translates the code

into one of the “mouse button pressed” codes at the bottom of Table 9-3. Furthermore, if you are not providing for any switch, desk accessory, device driver, or application-defined events, your program needn’t deal with codes between 9 and 15. That leaves only four event types to contend with: null, key-down, auto-key, and activate. If the program is entirely mouse-driven, and ignores key presses, the list is even shorter.

If your program has defined the TaskMaster task mask input with DC i4’\$01FFF’(as I suggested earlier), it must process only three task codes in the bottom portion of the list: system menu bar (17), content region (19), and close box (22). Menus will be discussed later, so you can disregard that topic for now. When the user presses the mouse button in the content region of the active window, TaskMaster stores a pointer to the window in the TaskData field of the task record and returns task code 19 on the stack. TaskMaster can’t process this event itself because the content region contains a picture the application has drawn, and the mouse may signify something in some applications.

When the user presses the mouse button in a window’s close box (or *go-away region*, in Apple’s documentation), TaskMaster stores a pointer to the window in the TaskData field of the task record and returns task code 22 on the stack. In general, your program should respond by calling HideWindow to make the window disappear. (How can the user make the window reappear? He or she normally does that by selecting from a menu.)

Tool Sets Required by TaskMaster

In the course of its work, TaskMaster checks for events from the Control Manager and the Menu Manager, so both of these tool sets must be active. I will discuss these managers in detail later, but for now I will simply present the start-up and shut-down calls that must be included in every program that calls TaskMaster (see Table 9-4).

These tool sets are RAM-based, so you must read them into memory by calling LoadTools. Furthermore, as you can see from the start-up calls, each manager requires one page in memory, so you must account for that when you call NewHandle.

An Example Window Program

I’ve covered a lot of window theory in this chapter, and you’re probably eager to try some of it. Clearly, it’s time to do some programming!

Table 9-4

<u>__CtlStartup</u>		Start the Control Manager
Call with:	PushWord <i>ProgramID</i> ;Program ID PushWord <i>DirectPageLoc</i> ;Loc. of 1-page work area <u>__CtlStartup</u>	
Result:	None	
<u>__CtlShutDown</u>		Shut down the Control Manager
Call with:	<u>__CtlShutDown</u>	
Result:	None	
<u>__MenuStartup</u>		Start the Menu Manager
Call with:	PushWord <i>ProgramID</i> ;Program ID PushWord <i>DirectPageLoc</i> ;Loc. of 1-page work area <u>__MenuStartup</u>	
Result:	None	
<u>__MenuShutDown</u>		Shut down the Menu Manager
Call with:	<u>__MenuShutDown</u>	
Result:	None	

The program listed in Example 9-1 (WINDOWS) displays three overlapping windows on a 640 mode screen, as shown in Figure 9-7. Each has a title bar, zoom and grow boxes, and both scroll bars — and all controls are operable. That is, you can drag a window by its title bar, make it zoom or grow, and scroll through its content region. WINDOWS keeps its windows on the screen until you press Esc.

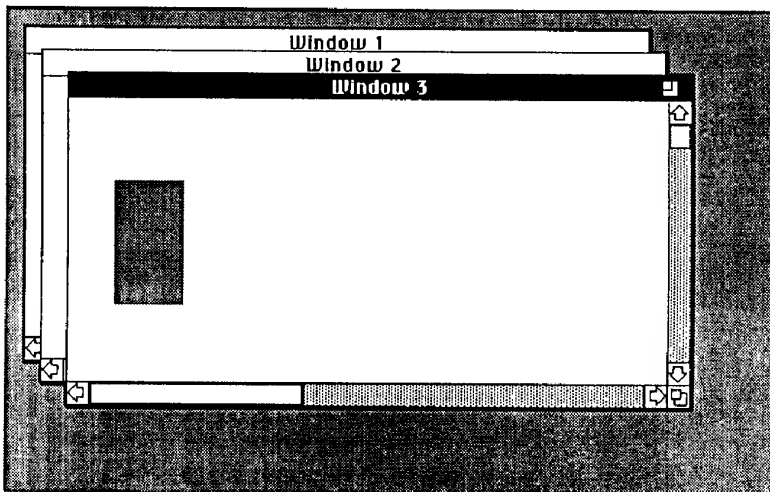


Figure 9-7

**Example 9-1**

; WINDOWS displays three windows and lets the user shuffle or
; drag them, or make them zoom or grow. To quit, press Esc.

```

        absaddr on
        MCOPY windows.macros

Windows      START
              using GlobalData

;-----
;
; Global equates used throughout the program.

ScreenMode   gequ $80                ;640 mode, no fill
MaxX          gequ 640                ;640 mode for Event Manager

              phk                      ;Set data bank to program
              plb                      ; bank to allow absolute addressing

              jsr InitStuff             ;Initialize everything
              bcs AllDone              ;Quit if initialization fails

              stz QuitFlag             ;Initialize the quit flag to 0
              jsr EventLoop            ;Let the user play

AllDone      anop                      ;All is done, shut down
              PushLong Win1Ptr         ;Close the windows
              _CloseWindow
              PushLong Win2Ptr
              _CloseWindow
              PushLong Win3Ptr
              _CloseWindow

              _DeskShutDown            ;Desk Manager
              _MenuShutDown            ;Menu Manager
              _WindShutDown            ;Window Manager
              _CtlShutDown             ;Control Manager
              _EMShutDown              ;Event Manager
              _QDShutDown              ;QuickDraw II
              _MTShutDown              ;Miscellaneous Tools

              PushWord MyID             ;Discard the program's handle
              _DisposeAll

              PushWord MyID
              _MMSHutdown              ;Memory Manager
              _TShutDown               ;Tool Locator

              _Quit QuitParams         ;Do a ProDOS Quit call
              brk $F0                  ;If it fails, break

              END

;*****
;
; Global Data
;
;*****

```

314 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

```

GlobalData    DATA

QuitParams    dc i'0'          ;Return to caller
               dc i'$4000'      ;Make program restartable in memory

ToolTable     dc i'NumTools'    ;No. of tool sets in table
               dc i'4,$0100'    ;QuickDraw
               dc i'5,$0100'    ;Desk Manager
               dc i'6,$0100'    ;Event Manager
               dc i'14,$0100'   ;Window Manager
               dc i'15,$0100'   ;Menu Manager
               dc i'16,$0100'   ;Control Manager
TTEnd         anop

TableSize     equ TTEnd-ToolTable-2
NumTools      equ TableSize/4

TaskRecord    anop             ;Buffer for task record
EventWhat     ds 2
EventMessage  ds 4
EventWhen     ds 4
EventWhere    ds 4
EventModifiers ds 2
TaskData      ds 4
TaskMask      dc i'$01FFF'     ;Make TaskMaster do all it can

Win1Ptr       ds 4             ;Window pointers
Win2Ptr       ds 4
Win3Ptr       ds 4

MyID          ds 2             ;This will hold the program's i.d.

VolNotFound   equ $45          ;ProDOS error

QuitFlag      ds 2             ;Quit flag
               END

;*****
;
; InitStuff
;
; Initializes tool sets, gets space in bank 0 for use as zero
; page by tool sets that need it, and ensures that RAM-based
; tools are in memory.
;
;*****

InitStuff     START
               using GlobalData

; Initialize the Tool Locator, Memory Manager, and Miscellaneous
; Tools.

               _TLStartup      ;Tool locator

               PushWord #0      ;Memory Manager
               _MMStartup

               pla              ;Memory Manager returns program's ID
               sta MyID

```

```

        _MTStartup                ;Misc. Tools
        ldx #3
        jsr PrepareToDie

; Get some space for the direct page we need. QuickDraw needs
; three pages and each Manager except Window needs one page.

        PushLong #0                ;Space for handle
        PushLong #$600            ;Six pages
        PushWord MyID             ;Owner
        PushWord #$C005           ;Locked, fixed, fixed bank
        PushLong #0              ;Location
        _NewHandle
        ldx #$FF
        jsr PrepareToDie

        pla                      ;Read handle and store in dp
        sta 0
        pla
        sta 2

        lda [0]                  ;Get dp location from handle
        sta 4                    ; and store at loc 4 of dp

; Initialize QuickDraw

        pha                      ;Use dp obtained from handle
        PushWord #ScreenMode      ;Mode = 640
        PushWord #160             ;Max size of scan line (in bytes)
        PushWord MyID
        _QDStartup
        ldx #4
        jsr PrepareToDie

; Initialize Event Manager

        lda 4                    ;dp to use = QD dp + $300
        clc
        adc #$300
        sta 4
        pha
        PushWord #20              ;Queue size
        PushWord #0               ;X clamp left
        PushWord #MaxX            ;X clamp right
        PushWord #0               ;Y clamp top
        PushWord #200             ;Y clamp bottom
        PushWord MyID
        _EMStartup
        ldx #6
        jsr PrepareToDie

;-----
;
; Load the RAM-based tools I need.

LoadAgain    PushLong #ToolTable
              _LoadTools
              bcc ToolsLoaded
              cmp #VolNotFound
              beq DoMount
              sec

```

316 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

```

        ldx #$FE
        jsr PrepareToDie

DoMount      anop
              jsr MountBootDisk
              cmp #1
              beq LoadAgain

              sec
              rts

; The tools have been loaded. Initialize the Window Manager, Control
; Manager, Menu Manager, and Desk Manager.

ToolsLoaded  anop
              PushWord MyID           ;Window Manager
              _WindStartup
              ldx #14
              jsr PrepareToDie

              PushLong #0             ;Prepare screen for windows
              _RefreshDesktop

              PushWord MyID           ;Control Manager
              lda 4                   ;dp to use = EM dp + $100
              clc
              adc #$100
              sta 4
              pha
              _CtlStartup
              ldx #16
              jsr PrepareToDie

              PushWord MyID           ;Menu Manager
              lda 4
              clc
              adc #$100
              pha
              _MenuStartup
              ldx #15
              jsr PrepareToDie

              _DeskStartup            ;Desk Manager

              jsr SetUpWindows        ;Show the windows
              _ShowCursor             ; and the mouse pointer

              jsr DrawWindows         ;Draw the window contents

              clc                     ;Clear the carry flag
              rts                     ; and return

END

; *****
;
; Set Up Windows
;
; Opens the three windows (but does not draw their contents).
;
; *****

```

```

SetUpWindows  START
               using GlobalData
               using WindowData

               PushLong #0                ;Window 1
               PushLong #Win1ParamBlock
               _NewWindow

               pla
               sta Win1Ptr
               pla
               sta Win1Ptr+2

               PushLong #0                ;Window 2
               PushLong #Win2ParamBlock
               _NewWindow

               pla
               sta Win2Ptr
               pla
               sta Win2Ptr+2

               PushLong #0                ;Window 3
               PushLong #Win3ParamBlock
               _NewWindow

               pla
               sta Win3Ptr
               pla
               sta Win3Ptr+2

               rts
               END

;*****
;
; Event Loop
;
; Display windows until user presses Esc.
;
;*****

EventLoop     START
               using GlobalData

Again         lda QuitFlag
               bne NoMore

               PushWord #0                ;Space for result
               PushWord #$FFFF            ;Accept any event
               PushLong #TaskRecord       ;Point to task record buffer
               _TaskMaster

               pla                        ;Is an event available?

               beq Again                  ; No. Continue polling
               asl a                      ; Yes. Double it
               tax                        ; and copy it into X.

               jsr (TaskTable,x)          ;Execute the event's routine,
               bra Again                  ; then resume polling

NoMore        rts

```

318 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

```

TaskTable      anop                      ;Event Manager events
                dc i'Ignore'             ; 0 null
                dc i'Ignore'             ; 1 mouse down
                dc i'Ignore'             ; 2 mouse up
                dc i'DoKeyDown'           ; 3 key down
                dc i'Ignore'             ; 4 undefined
                dc i'Ignore'             ; 5 auto-key
                dc i'Ignore'             ; 6 update
                dc i'Ignore'             ; 7 undefined
                dc i'Ignore'             ; 8 activate
                dc i'Ignore'             ; 9 switch
                dc i'Ignore'             ; 10 desk acc
                dc i'Ignore'             ; 11 device driver
                dc i'Ignore'             ; 12 ap
                dc i'Ignore'             ; 13 ap
                dc i'Ignore'             ; 14 ap
                dc i'Ignore'             ; 15 ap
                dc i'Ignore'             ; 16 in desktop
                dc i'Ignore'             ; 17 in system menu bar
                dc i'Ignore'             ; 18 in system window
                dc i'Ignore'             ; 19 in window content region
                dc i'Ignore'             ; 20 in drag region (title bar)
                dc i'Ignore'             ; 21 in grow box
                dc i'Ignore'             ; 22 in go-away region (close box)
                dc i'Ignore'             ; 23 in zoom box
                dc i'Ignore'             ; 24 in information bar
                dc i'Ignore'             ; 25 in right scroll bar
                dc i'Ignore'             ; 26 in bottom scroll bar
                dc i'Ignore'             ; 27 in frame
                dc i'Ignore'             ; 28 in drop region
                END

```

```

;*****
;
; Ignore
;
; This do-nothing subroutine returns to the event loop for
; events we don't want.
;
;*****

```

```

Ignore      START
            rts
            END

```

```

;*****
;
; DoKeyDown
;
; If user pressed Esc, DoKeyDown sets the QuitFlag to 1. It
; ignores any other key.
;
;*****

```

```

DoKeyDown    START
            using GlobalData

            lda EventMessage          ;Read the ASCII character
            cmp #27                   ;Is it Esc?
            bne Exit

```



```

Exit          sta QuitFlag          ; Yes. Quit
              rts

              END

;*****
;
; Window Data
;
;*****

WindowData    DATA

Win1ParamBlock anop
    dc        i'Win1End-Win1ParamBlock'
    dc        i2'%1001110110000000' Everything but close box & info bar
    dc        i4'Win1Title'         Pointer to window's title
    dc        i4'0'                 Reserved (wRefCon)
    dc        i2'26,0,190,620'       Zoomed content region
    dc        i4'0'                 Default color table
    dc        i2'0'                 Vertical origin
    dc        i2'0'                 Horizontal origin
    dc        i2'200'               Data area height
    dc        i2'640'               Data area width
    dc        i2'200'               Max grow height
    dc        i2'640'               Max grow width
    dc        i2'4'                 Number of pixels to scroll vertically.
    dc        i2'16'               Number of pixels to scroll
horizontally.
    dc        i2'40'               Number of pixels to page vertically.
    dc        i2'160'             Number of pixels to page horizontally.
    dc        i4'0'               Infomation bar text string.
    dc        i2'0'               Info bar height
    dc        i4'0'               Routine to draw shape (none, standard)
    dc        i4'0'               Routine to draw info. bar.
    dc        i4'DrawWin1'         Routine to draw content.
    dc        i'40,20,100,360'      Size/pos of content
    dc        i4'-1'               Window's order (-1 means topmost)
    dc        i4'0'               Window Manager allocates wind. record
Win1End       anop

Win2ParamBlock anop
    dc        i'Win2End-Win2ParamBlock'
    dc        i2'%1001110110000000' Everything but close box & info bar
    dc        i4'Win2Title'         Pointer to window's title
    dc        i4'0'                 Reserved (wRefCon)
    dc        i2'26,0,190,620'       Zoomed content region
    dc        i4'0'                 Default color table
    dc        i2'0'                 Vertical origin
    dc        i2'0'                 Horizontal origin
    dc        i2'200'               Data area height
    dc        i2'640'               Data area width
    dc        i2'200'               Max grow height
    dc        i2'640'               Max grow width
    dc        i2'4'                 Number of pixels to scroll vertically.
    dc        i2'16'               Number of pixels to scroll
horizontally.
    dc        i2'40'               Number of pixels to page vertically.

```

```

        dc      i2'160'          Number of pixels to page horizontally.
        dc      i4'0'            Infomation bar text string.
        dc      i2'0'            Info bar height
        dc      i4'0'            Routine to draw shape (none, standard)
        dc      i4'0'            Routine to draw info. bar.
        dc      i4'DrawWin2'      Routine to draw content.
        dc      i'50,30,110,380'  Size/pos of content
        dc      i4'-1'           Window's order (-1 means topmost)
        dc      i4'0'            Window Manager allocates wind. record
Win2End  anop

Win3ParamBlock anop
        dc      i'Win3End-Win3ParamBlock'
        dc      i2'$1001110110000000' Everything but close box & info bar
        dc      i4'Win3Title'     Pointer to window's title
        dc      i4'0'            Reserved (wRefCon)
        dc      i2'26,0,190,620'  Zoomed content region
        dc      i4'0'            Default color table
        dc      i2'0'            Vertical origin
        dc      i2'0'            Horizontal origin
        dc      i2'200'           Data area height
        dc      i2'640'           Data area width
        dc      i2'200'           Max grow height
        dc      i2'640'           Max grow width
        dc      i2'4'            Number of pixels to scroll vertically.
        dc      i2'16'           Number of pixels to scroll
horizontally.
        dc      i2'40'           Number of pixels to page vertically.
        dc      i2'160'          Number of pixels to page horizontally.
        dc      i4'0'            Infomation bar text string.
        dc      i2'0'            Info bar height
        dc      i4'0'            Routine to draw shape (none, standard)
        dc      i4'0'            Routine to draw info. bar.
        dc      i4'DrawWin3'      Routine to draw content.
        dc      i'60,40,120,400'  Size/pos of content
        dc      i4'-1'           Window's order (-1 means topmost)
        dc      i4'0'            Window Manager allocates wind. record
Win3End  anop

Win1Title  str 'Window 1'
Win2Title  str 'Window 2'
Win3Title  str 'Window 3'

```

```

PenPat1  dc h'11 11 11 11 11 11 11 11' ;Dithered blue pen pattern
        dc h'11 11 11 11 11 11 11 11'
        dc h'11 11 11 11 11 11 11 11'
        dc h'11 11 11 11 11 11 11 11'

Rect      dc i'20,20,40,40'          ;Rectangle painted in windows

```

END

```

;*****
;
; Draw Windows
;
; Fills in the content region of each window, working rearward.
;
;*****

```

```
DrawWindows    START
    using GlobalData
```

```
    PushLong Win3Ptr          ;Window 3
    _SetPort
```

```
    jsl DrawWin3
```

```
    PushLong Win3Ptr
    _ShowWindow
```

```
    PushLong Win2Ptr          ;Window 2
    _SetPort
```

```
    jsl DrawWin2
```

```
    PushLong Win2Ptr
    _ShowWindow
```

```
    PushLong Win1Ptr          ;Window 1
    SetPort
```

```
    jsl DrawWin1
```

```
    PushLong Win1Ptr
    _ShowWindow
```

```
    rts
    END
```

```
;*****
;
; Draw Window 1
;
; Draws the contents of Window 1--a blue box on a white
; background. Called by DrawWindows initially and by the Window
; Manager when it updates the window.
;
;*****
```

```
DrawWin1 START
    using WindowData
```

```
    PushWord #3                ;Background is white
    _SetSolidBackPat
```

```
    PushLong #PenPat1          ;Set pen pattern to blue
    _SetPenPat
```

```
    PushLong #Rect             ; and paint the rectangle
    _PaintRect
```

```
    rtl                        ;RTL required for update events
    END
```

322 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

```
;*****
;
; Draw Window 2
;
; Draws the contents of Window 2--a white box on a green
; background. Called by DrawWindows initially and by the Window
; Manager when it updates the window.
;
;*****

DrawWin2 START
    using WindowData

    PushWord #2                ;Background is green
    _SetSolidBackPat

    PushWord #3                ;Set pen pattern to white
    _SetSolidPenPat

    PushLong #Rect             ; and paint the rectangle
    _PaintRect

    rtl                        ;RTL required for update events
    END

;*****
;
; Draw Window 3
;
; Draws the contents of Window 3--a green box on a red
; background. Called by DrawWindows initially and by the Window
; Manager when it updates the window.
;
;*****

DrawWin3 START
    using WindowData

    PushWord #1                _SetSolidBackPat

    PushWord #2                ;Set pen pattern to green
    _SetSolidPenPat

    PushLong #Rect             ; and paint the rectangle
    _PaintRect

    rtl                        ;RTL required for update events
    END

*****
*
* Prepare To Die
*
* Dies if carry is set.
*
* Error number to display is in X register.
* Assumes that Miscellaneous Tools is active.
*
*****
```

```

PrepareToDie  START
               bcs RealDeath      ;Carry = 1?
               rts                ; No. Return to caller

RealDeath     phx                  ; Yes. Goodbye, program.
               PushLong #DeathMsg
               _SysFailMgr

DeathMsg      str 'Could not handle error '

               END

;*****
;
; Mount Boot Disk
;
; Tells the user to insert the disk that contains the RAM-based
; tools.
;
;*****

MountBootDisk START

               _Set_Prefix SetPrefixParams
               _Get_Prefix GetPrefixParams

               PushWord #0          ;Space for result
               PushWord #195        ;Column position for dialog box
               PushWord #30         ;Row position for dialog box
               PushLong #PromptStr  ;Prompt at top of dialog box
               PushLong #VolStr     ;Volume name string
               PushLong #OKStr      ;String in Button 1
               PushLong #CancelStr  ;String in Button 2
               _TLMountVolume

               pla                  ;Obtain the button number
               rts                ; and return to caller

PromptStr     str 'Please insert the disk.'
VolStr        ds 16
OKStr        str 'OK'
CancelStr     str 'Shutdown'

GetPrefixParams dc i'7'
               dc i4'VolStr'

SetPrefixParams dc i'7'
               dc i4'BootStr'

BootStr       str '*/'

               END

```

I *could* have displayed a complex pixel image in each window, but that would have made the program longer and more difficult to understand. Instead, I used QuickDraw to draw a colored box in each one. Window 1

has a blue box on a white background, Window 2 has a white box on a green background, and Window 3 has a green box on a red background. This may not be very creative on my part, but it serves to keep the program to minimal length. Once you get WINDOWS up and running, you may want to modify it by inserting QuickDraw calls of your own.

The WINDOWS program is quite long, but if you understand the earlier programs in this book and the material in this chapter, you shouldn't have any problem with it. Here is how WINDOWS operates.

To begin, the Windows segment calls `InitStuff` to initialize the tools and set up the windows. `InitStuff` starts the same tools as the earlier `SHOWTEXT` program (Example 8-3), but it also starts the Window Manager, the Menu Manager, and the Control Manager. (The Menu and Control Managers must be active for TaskMaster to operate.) When the tools have been initialized, `InitStuff` calls `SetUpWindows` to display the windows.

`SetUpWindows` performs three `NewWindow` calls, one for each window, based on parameter tables in the `WindowData` segment. By examining the `wFrame` parameter (the second entry in each table), you will notice that I have made each window invisible and have given it a movable title bar, grow and zoom boxes, and scroll bars. I have not provided an information bar because it is unnecessary. I haven't provided a close box either, because I have not yet described how to reinstate a window once it has been hidden. (That's coming in Chapter 10.)

Each window parameter table also includes a pointer to a subroutine (`DrawWin1`, `2`, or `3`) that the Window Manager will use to draw the contents of an inactive window when you activate it — that is, when you press the mouse button anywhere in it. Each `DrawWin` subroutine sets the background and pen patterns (i.e., their colors), then paints a rectangle.

`DrawWin2` and `DrawWin3` use `SetSolidPenPat` to set the color, because they draw with colors from the first four entries of the 640 mode color table. `DrawWin1` must use `SetPenPat`, because it draws with blue, which can only be obtained through dithering. (If you specify color 5, blue in the 640 mode table, Quickdraw will ignore the 1 in bit 3 and draw the rectangle using color 1, red.)

Note that each `DrawWin` subroutine ends with an *rtl* (Return from Subroutine Long) rather than an *rts* (Return from Subroutine). They must be "long" because they are called by the Window Manager, which may be in a different bank than the program.

Upon return from `SetUpWindows`, `InitStuff` displays the mouse pointer with a QuickDraw `ShowCursor` call, then executes a `DrawWindows` subroutine to draw the content region of each window. To do this, `DrawWindows`

makes the window's GrafPort active by doing a SetPort, then calls the DrawWin subroutine to do the actual drawing.

Upon return from DrawWindows, InitStuff returns control to the Windows segment. Windows then calls PlayWithIt, the user's evnt loop. PlayWithIt is simply a loop that polls TaskMaster continually until the user presses Esc. I could have used GetNextEvent here, but why bother? TaskMaster monitors the mouse position and tells the program when the button has been pressed in a window control. With GetNextEvent, the program would have to do all that.

When the user finally presses Esc, PlayWithIt exits to the Windows segment. The only remaining job is to close the windows and shut down the tools.

CHAPTER 10

Menus

In the early days of personal computers, users could only interact with a program by pressing keys. The more polished programs presented a list, or *menu*, of the available choices; programs of the cruder variety forced the user to remember keyboard commands. All this changed when the mouse entered the scene — now the user could point at his or her menu choice and press the mouse button to select an item.

Because the Apple IIGS comes equipped with a mouse, your programs should take advantage of it whenever possible; that is, the program should display a *menu bar* at the top of the screen, and let the user select from menu items or “pull-down” submenus by pointing and pressing the mouse button. To use menus in your program, you need the services of the *Menu Manager*.

Menu Bars and Pull-Down Menus

A menu bar is a rectangular strip that displays a list of commands the user can activate by pressing the mouse button. Some of these commands may cause some predefined task to be performed. (For example, selecting “Cut” from the menu bar of a word-processing program generally removes a block of highlighted text from the screen.) More often, however, menu bar

commands are actually menu titles. Pressing the mouse button on a title causes a menu to drop down onto the screen. From now on, I'll refer to items on a menu bar as "titles," but be aware that they may represent menuless commands.

The System Menu Bar

Applications that use pull-down menus must present the titles of those menus in a *system menu bar* at the top of the screen. Menu titles will vary between programs, but the *Apple Human Interface Guidelines* has defined two of them, *File* and *Edit*, that commonly appear in application programs.

According to the *Guidelines*, selecting "File" should produce a menu that lets the user perform simple file-related operations. Typical commands in the File menu include New (open a new, untitled document), Save (store the document on disk), and Print. The File menu should also include the Quit command, which lets the user exit from the program.

Selecting the second standard menu title, "Edit," should produce a menu that provides commands for operating on objects. In a text-based program, for instance, the Edit menu may include Undo, Cut, Copy, and Paste commands.

The system menu bar may also include a special graphics object: the shape of an apple. (Perhaps I should say *the* apple, because this particular one is decked out in the logo colors of Apple Computer, Inc.) Selecting the apple calls up a menu of the available new desk accessories (NDAs). Usually, this menu also provides an *About* command that the user can select to learn more about using the program.

For example, in a program called MoneyMaker, the About command may read "About MoneyMaker . . ." Selecting it brings up an on-screen window with, perhaps, a copyright notice, the name and address of the program's publisher, instructions for using the program, and anything else the application wants to display. Of course, the window must also have an *OK* button that the user can press to make the window disappear.

You're probably wondering about this "button" business. After all, I haven't mentioned this feature before. There's a good reason for that: the About window is not an ordinary Window Manager window; it is a *dialog box* that has been produced by (you guessed it) the Dialog Manager.

Figure 10-1 shows a system menu bar equipped with the apple, File and Edit titles, and View and Special titles that produce menus pertinent to the menu bar's application program.

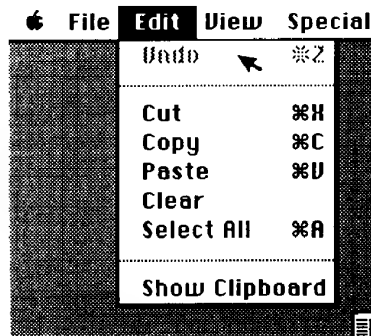


Figure 10-1

Pull-Down Menus

You have probably used pull-down menus already, and know how they work, so you may be tempted to skip this section. Don't! Although the way pull-down menus work is transparent to the user, and everything seems to take place automatically, the application program must do a lot of operations behind the scenes to make all this happen. For this reason, it's worthwhile to spend a little time investigating how pull-down menus operate. For the time being, I will describe their operation from a user's viewpoint. Later, I'll discuss them from a programmer's viewpoint.

When you point to a menu title and press the mouse button, the title text changes color and the title's menu drops down into a small window on the screen. Then, as you move the pointer down the menu (with the mouse button still down), each command is highlighted as the pointer reaches it; see Figure 10-2. Finally, when you reach the command you want, you release the mouse button. This makes the command name blink briefly and the menu disappear. When the program has finished executing the command, the menu title returns to its original color.

With all this happening, you may expect the programming for pull-down menus to be difficult. Relax. The Menu Manager (bless its heart) does much of the work, so the programming is rather easy (see later in this chapter).

Enabled and Disabled Menus

Sometimes you may want to prevent the user from selecting a certain menu or an item within it. For example, you may not want a game player to "Start a New Game" before he or she has told your program to "End This Game."

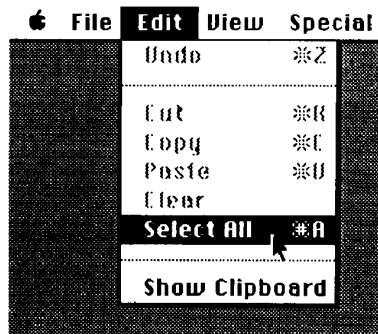


Figure 10-2

To prevent users from selecting an invalid menu title or menu item, you can disable it. When a menu title is disabled, it appears rather dim and fuzzy. The user can still pull its menu down, but every item will be dim and none of them will be selectable. Similarly, any disabled items in an enabled menu are also unselectable. Figure 10-3 shows an Edit menu with a disabled Undo command.

Menu Items

So far, I have referred to menu items as commands (words or phrases) the user can select by highlighting and releasing the mouse button — and indeed, that's what they usually are. However, you can also create menus that have a few variations, such as the one shown in Figure 10-4.

For example, you can divide groups of commands by function, by putting a dimmed line between them. A dividing line is itself a menu item, but since it is dimmed and disabled, the user can never select it. To him or her, it is simply a graphics object. The menu in Figure 10-4 uses dividing lines to separate the related commands Cut, Copy, and Paste from the commands Undo and Draw.

You can also precede a menu item with a *mark* (any character) to indicate some sort of status. For example, the menu in Figure 10-4 shows a crosshatch in front of the Draw command, to signify that the Draw feature is in effect. This program might also mark the Cut or Copy command when the user has cut or copied material but has not yet pasted it. The mark, then,

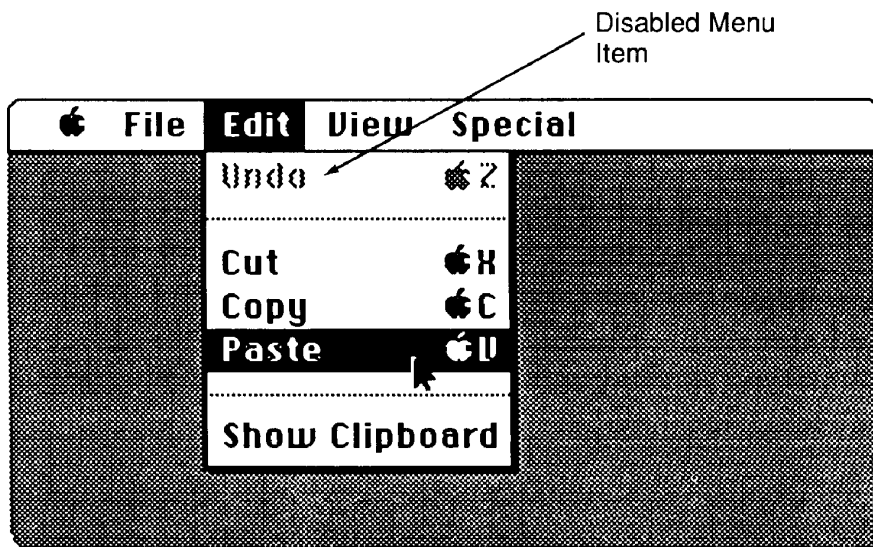


Figure 10-3

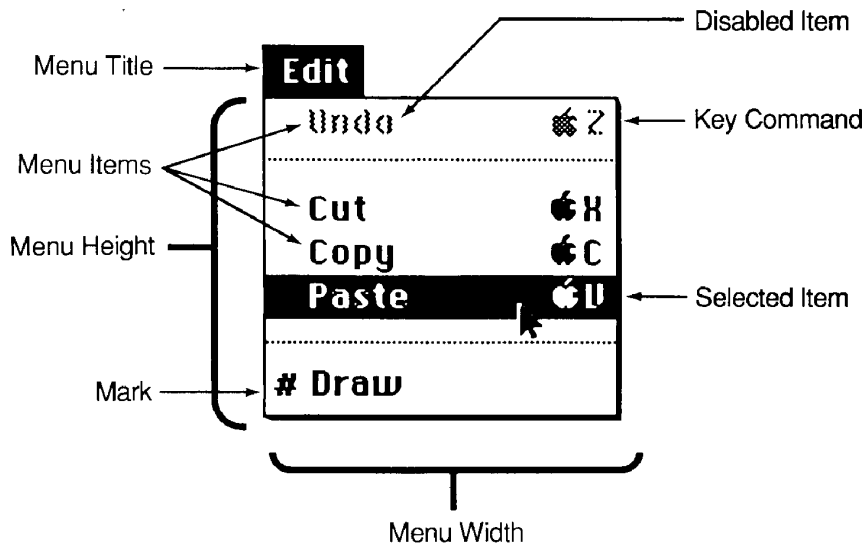


Figure 10-4

serves as a simple and effective way of telling the user about something without displaying a special message.

If there is a key equivalent for a menu item, your program can show it to the right of the command name. Generally, you should make the user hold down the OpenApple key when he or she presses the character key. This is indicated by showing an apple symbol ahead of the character. For example, the menu in the figure shows that the user can give an Undo, Cut, Copy, or Paste command by pressing OpenApple and Z, X, C, or V.

The letter keys for the Undo, Cut, Copy, and Paste commands weren't selected at random. They are the ones that Apple Computer recommends for use in *every* program that includes these operations. There are five more "standard" key combinations, for a total of nine. Here's the complete list:

Apple Menu

OpenApple-?	Help
-------------	------

File Menu

OpenApple-N	New
OpenApple-O	Open
OpenApple-S	Save
OpenApple-Q	Quit

Edit Menu

OpenApple-Z	Undo
OpenApple-X	Cut
OpenApple-C	Copy
OpenApple-V	Paste

Having a keyboard equivalent for Quit lets the user exit without selecting from a menu.

Creating Menu Bars and Menus

To equip your program with one or more menus, you first create the title bar, then insert the menu in it. To do this:

1. Initialize the Menu Manager with a MenuStartup call.

2. Define each menu and the items in it with a `NewMenu` call.
3. Add the menu to the system menu bar with an `InsertMenu` call.
4. Calculate the sizes of the menu bar and the menus with a `FixMenu-Bar` call.
5. Display the menu bar and the menu titles with a `DrawMenuBar` call.

At the end of this sequence, your menu bar is on the screen and the menus are in place (although they're invisible), ready to pull down and use. Table 10-1 summarizes these tool calls, in the order of their use.

Since I have just explained what these calls do, and the order in which they should appear in your programs, I won't bother describing them individually. However, I must explain the so-called menu/item line list that `NewMenu` requires.

The Menu/Item Line List

The `NewMenu` tool call sets up a menu title and the contents of items in the menu based on a list in your program. The length of this *menu/item line list* depends on the length of the menu.

Every menu/item line list is at least two lines long, where the first line defines the contents of the menu title (or command, in this case) and the second line contains an end-of-list, or *termination*, character. If the word or phrase in the menu bar is actually a menu title, the list must have one additional line for each item in the menu. Thus, the menu/item line list for a five-item menu must have seven lines: a title line, five item lines, and a termination line.

The title and item lines can contain up to five components, as follows:

1. A *start character*. You can use any characters you want, as long as the title and item characters are different.

I like to start title lines with a right angle bracket (>) and item lines with a space. The > symbol makes the list's title line easy to spot in a program listing (or search for using the editor), while the space character gives item lines a cleaner look than they would have with a "visible" character.

2. A one-character *place marker* in which `NewMenu` will store the length of the text string for the menu title or item name. I generally use L (for Length) for the marker, but any character will do. After all, the place marker doesn't *do* anything other than reserve space.

Table 10-1

<u>MenuStartup</u>		Start the Menu Manager
Call with:	PushWord <i>ProgramID</i> PushWord <i>DirectPageLoc</i> __MenuStartup	;Program ID ;Loc. of 1-page work area
Result:	None	
<u>NewMenu</u>		Create a new menu
Call with:	PushLong #0 PushLong <i>MenuStringPtr</i> __NewMenu	;Space for result ;Pointer to a menu/item line list
Result:	Handle of menu (long word); zero if an error occurred.	
Note:	See text for description of the menu/item line list.	
<u>InsertMenu</u>		Insert a menu in the current menu bar
Call with:	PushLong <i>AddMenu</i> PushWord #0 __InsertMenu	;Menu handle ;Insert at front of menu bar
Result:	None	
Note:	If you call InsertMenu immediately after NewMenu, the menu handle is already on the stack, and you can use the shorter form: PushWord #0 __InsertMenu	
<u>FixMenuBar</u>		Calculate sizes of menu bar and menus
Call with:	PushWord #0 __FixMenuBar	;Space for result
Result:	Height of menu bar (word).	
Note:	Normally you don't care about the height of the menu bar, and should follow FixMenuBar with a PLA instruction.	
<u>DrawMenuBar</u>		Display the completed menu bar
Call with:	__DrawMenuBar	
Result:	None	
<u>MenuShutDown</u>		Shut down the Menu Manager
Call with:	__MenuShutDown	
Result:	None	

3. The *text* for the menu title or item name, followed by a backslash (\) character.
4. The character N followed by the *identification number* of the menu or item. Working from left to right along the menu bar, menus are numbered from 1 to 255, while items in the program are numbered from 256 to 65534(!).
5. Optional *menu modifiers* (to be discussed shortly).

6. A *line termination character*; either a Return (decimal 13) or a Null (decimal 0).

The termination line at the end of the list contains only a termination character. This can be any character except the one you use to start item lines. A period (.) is a reasonable choice here.

For example, the following menu/item line list defines a menu that has only one item:

```
Menu1  dc c'>L File \N1',il'13' ;Title line
        dc c' LQuit\N256',il'13'   ;Item line
        dc c'.'
```

The N1 on the first line indicates that this menu title is to be inserted at the leftmost position on the spacebar.

Menu Modifiers

As just mentioned, the identification number on a title or item line may be followed by one or more menu modifiers. As Table 10-2 shows, there are modifiers that let you specify an item's text style (emboldened, underlined, or italicized), and show a marking symbol to the left of the name and a key command to the right of it. Note that you can use any of the modifiers to define an item, but only D and X to define a title.

If a menu item has an equivalent *key command*, you should remind the user of it by displaying the key or key combination to the right of the item name. To specify a key command for an item, include the * modifier in its

Table 10-2

Modifier	Action
*Kk	Display a key equivalent for the menu item, where <i>K</i> is the primary character and <i>k</i> is the secondary character.
B	Display the menu item text in bold type.
C <i>m</i>	Precede (mark) the menu item text with an <i>m</i> character.
D	Disable the menu item or title (i.e., make it dim).
I	Display the menu item text in italics.
U	Underscore the menu item text.
V	Display a dividing line beneath this item.
X	Use color replace highlighting

definition. This makes the Menu Manager display an apple symbol to the left of your primary character. The apple tells the user that to activate this command from the keyboard, he or she must press the OpenApple key as well as the character key.

Generally, the primary character is an uppercase letter, so the secondary character is its lowercase counterpart. For example, entering **Rr** lets the user press either OpenApple-Shift-R or OpenApple-R to access the menu item. (The terms *primary* and *secondary* are somewhat misleading here. While the primary character is the one that is displayed on the screen, all but rank novices will press OpenApple key, instead of OpenApple-Shift key. In this case, then, the so-called secondary character is actually the one that will be used most often.)

If the key command has no secondary character (e.g., it is a number), you must enter a space after the primary character. For instance, “*2 ” establishes OpenApple-2 as the only valid key command for this item.

I will discuss the *X* modifier shortly. *B*, *I*, and *U* are self-explanatory; they embolden, italicize, or underline an item. Underlining is useful for saving space in a long menu, because it doesn't take up a line of its own. The Menu Manager provides `GetItemStyle` and `SetItemStyle` tool calls that let you read and change these attributes from your program.

C lets you mark an item name with a character of your choice, where crosshatch (#) and asterisk (*) are the most likely candidates. The Menu Manager provides a `SetItemMark` call that lets you mark or unmark an item name from within a program. It also has a `CheckItem` call that stamps an item with a check mark, the symbol the Control Panel shows to indicate a default setting.

D makes an item unselectable and shows it dimmed. You can enable and undim an item from your program by calling the Menu Manager's `EnableItem` tool. Similarly, you can disable an item by calling `DisableItem`.

V inserts a dividing line between this item and the next one. Thus, *V* is similar to *U* (underline), except it gives the menu a more orderly appearance; *U* tends to jam items together vertically.

Figure 10-5 shows a menu that has the features I have described so far. Its menu/item line list would have the form:

```
Menu4 dc c' >LText\N4',il'13'           ;Title line
      dc c' LUndo\N264NV',il'13'         ;Dimmed,
                                           ; dividing line
      dc c' LLeft\N265C#*L1',il'13'      ;Marked with #,
                                           ; L keys
```

Text	
Undo	
#Left	⌘ L
Centered	⌘ C
Right	⌘ R
Bold	⌘ B
<i>Italic</i>	⌘ I

Figure 10-5

```

dc c' LCentered\N266*Cc',il'13' ;C key command
dc c' LRight\N267U*Rr',il'13'   ;Underlined,
                                ; R keys
dc c' LBold\N268B*Bb',il'13'   ;Bold, B keys
dc c' LItalics\N269I*Ii',il'13' ;Italics,
                                ; I keys
dc c' .'                        ;Termination
                                ; line

```

This list assumes that the Text menu is the fourth menu in the program (it is numbered N4) and that the preceding menus had a total of eight items (numbered 256 through 263).

You can also display the multicolored Apple symbol as a menu title; simply enter @ for the title and put X after the menu number. By convention, if you use an Apple menu, make it the first one on the menu bar. The following directives define an Apple menu with one item, "About this program . . ."

```

Menu1 dc c'>L\N1X',il'13'      ;Apple menu title line
      dc c' LAbout this program...\N256',il'13'
                                      ;Item line
      dc c' .'                  ;Termination line

```

You can also specify a disabled dividing line as an item, by entering a hyphen (-) for the title and a D after the item number, as in:

```
dc c' L-\N277D' ,il'13' ;Dividing line
```

Responding to Menu Events

It's possible to use menus with just the Event Manager and the Menu Manager in place, and to poll for mouse-up and mouse-down events with the `GetNextEvent` call. However, as an Event Manager tool, `GetNextEvent` only records the mouse's location when the event occurred; it neither knows nor cares whether that position is relevant. Thus, with `GetNextEvent`, your program must do all the work of relating a mouse event to the pointer location. It's much easier to activate the Window Manager — even if your application doesn't use windows — and let its `TaskMaster` keep track of the mouse.

`GetNextEvent` also falls short in processing key commands. No matter which key the user presses (and regardless of whether OpenApple is also pressed), `GetNextEvent` generates a key-down event and makes your program decipher the key code. By contrast, `TaskMaster` intercepts Open-Apple key commands automatically and, with help from the Menu Manager, executes the routines they represent. Your program never gets involved with carrying out key commands.

Mouse Events

When the user presses the mouse button in the system menu bar, `TaskMaster` executes the Menu Manager's *MenuSelect* routine. If the mouse pointer is not in a menu title, *MenuSelect* tells `TaskMaster` to ignore it and `TaskMaster` returns 0 (the null event) on the stack, as if the button had never been pressed.

However, if the mouse pointer *is* in a menu title when the user presses the button, *MenuSelect* highlights the title and pulls down its menu (if any). Then it puts the title's ID number in the high word of the task record's `TaskData` field and 0 in `TaskData`'s low word. Of course, *MenuSelect* tells `TaskMaster` about the selection, but your program never hears about it. And why should it? After all, the user has simply moved the pointer to a selectable title; he or she has not yet *chosen* anything. To choose a menu item, you must *release* the mouse button while the item is selected (highlighted).

Hence, while the mouse button is being held down, `TaskMaster` calls *MenuSelect* repeatedly, but returns nulls to your program. The user's selections are a private matter between these two tools; the program needn't know about them.

If the user moves the pointer off the title and down to the first menu item, *MenuSelect* switches the highlighting from the title to the item and

stores the menu's ID in the high word (as before) and the item's ID in the low word. In fact, once the mouse button is down, MenuSelect doggedly monitors the pointer location. If the user moves the pointer to a new item, MenuSelect switches the highlighting and updates TaskData.

What happens when the user finally releases the button depends on where the mouse pointer is located? If the user has moved the pointer completely off the menu before he or she releases the button, MenuSelect says "Okay, false alarm," and takes the menu off the screen. It's then up to TaskMaster to inform your program where the button was released — in the system window, in the content region of an application window, or whatever. If the user releases the mouse button in a highlighted (selected) title or menu item, TaskMaster presents your program with task code 17, indicating "in system menu bar."

Responding to Mouse Events

Upon obtaining task code 17 from the stack, the program's event loop should use it as an index into a task table, just as the WINDOWS program in Chapter 9 does. Entry 17 in the table should point to a subroutine called, say, DoMenu that processes menu selections. DoMenu's job is to read what MenuSelect has stored in TaskData, act on it, then turn the menu's highlighting off and return to the event loop.

The content of DoMenu depends on how your program is using the menu bar. If any title is a stand-alone command (i.e., a title with no menu), DoMenu should read its ID number from the high word of TaskData and use that number to index into a table of command-processing subroutines. If your menu bar has four titles, DoMenu may look like Example 10-1.

On the other hand, if each of the words or phrases on the menu bar represents a menu title, your DoMenu subroutine can ignore the menu IDs in the high word of TaskData and work directly with the item IDs in TaskData's low word. Because item ID numbers start with 256, DoMenu should subtract 256 from the number (or simply clear bit 8) and use the result to index into your menu table — or item table, in this case. Example 10-2 shows the instructions you need here.

Key Events

Just as the TaskMaster works in conjunction with MenuSelect to keep track of mouse events for menus, it works with another Menu Manager routine, MenuKey, to check key-down events against key commands within the menus.

Example 10-1

```

DoMenu    START
          using GlobalData

          lda TaskData+2          ;Read the menu number
          asl a                  ; and double it
          tax

          jsr (MenuTable,x)       ;Go process the selection

          PushWord #0             ;Unhighlight the menu
          PushWord TaskData+2
          _HiliteMenu
          rts                    ; and return

MenuTable dc i'Ignore'           ;There is no Menu 0
          dc i'DoCommand1'
          dc i'DoCommand2'
          dc i'DoCommand3'
          dc i'DoCommand4'

          END

```

Example 10-2

```

DoMenu    START
          using GlobalData

          lda TaskData           ;Read the item number
          and #$00FF            ;Strip off the "256" bit
          asl a                  ; and double the result
          tax

          jsr (ItemTable,x)      ;Go process the selection

          PushWord #0            ;Unhighlight the menu
          PushWord TaskData+2
          _HiliteMenu
          rts                    ; and return

ItemTable dc i'DoItem1'
          dc i'DoItem2'
          dc i'DoItem3'
          dc i'DoItem4'
          ::
          ::

          END

```

When the user presses a key, TaskMaster does two things. First it stores the key's numeric code and the up/down state of the modifier keys (Control, Shift, Open-Apple) in the task record, then it sends this key information to

the MenuKey routine. MenuKey looks at this information to see whether the Open-Apple key was pressed. If not, it tells TaskMaster, “This has nothing to do with menu key commands,” and TaskMaster sends a key-down event to your program.

Otherwise, if the user pressed OpenApple as well as a character key, MenuKey checks whether that particular combination corresponds to a key command in any menu item. If MenuKey finds a matching item, it tells TaskMaster. TaskMaster, in turn, stores the menu’s ID number in TaskData and presents task code 17 (“in system menu bar”) to your program. The program doesn’t realize that a key command caused the event; it executes the item’s subroutine just as though the user had selected the item with the mouse.

This working partnership between TaskMaster and MenuKey lets your programs disregard key commands entirely. Unless you’re using keys for some other purpose, you can put an “ignore” in the task record’s key-down event entry.

Providing for New Desk Accessories

If your program has any menus at all, it should allow the user to call up any installed new desk accessories (NDAs). NDAs are kept in the DESK.ACCS subdirectory of the boot disk’s SYSTEM directory. The tool that provides for NDAs is one provided by the Desk Manager — FixAppleMenu. Details are:

```
_FixAppleMenu  Insert list of NDAs in the specified menu
                Call with:  PushWord MenuNumber      ;Menu number
                        _FixAppleMenu
                Result:  None
```

By convention, you should put the list of NDAs in the Apple menu. Since that is the first menu on the menu bar, your input to FixAppleMenu should be *PushWord #1*. The FixAppleMenu call belongs immediately before your FixMenuBar call.

An Example Program that Provides Menus

Example 10-3 lists a program called MENUS that’s an enhanced version of the WINDOWS program in Chapter 9. MENUS displays the same three

windows as before (although each now has a close box), but it also provides menus. The menu bar at the top of the screen has an Apple menu symbol and the titles File and Windows.

The Apple menu has only one item, *About this program*, but this item is disabled (dimmed). I will activate it in a later program.

The File menu also has only one item, Quit, which lets the user exit. Quit has the equivalent key command OpenApple-Q. The third menu, Windows, lets the user select a window and bring it to the front of the stack. This menu's main purpose is to let the user reinstate a window that he or she has closed previously.

By glancing through the listing, you will notice relatively few differences between WINDOWS and MENUS. The InitStuff segment now includes NewMenu and InsertMenu calls to build the menus, a Desk Manager FixAppleMenu call to install the list of NDAs in the Apple menu, a FixMenuBar call to size the menus, and a DrawMenuBar call to display the completed system menu bar. The task table in the EventLoop now has an "Ignore" for key-down events (KeyMenu will handle them automatically) and pointers to DoMenu and DoClose subroutines for system menu bar and go-away events (events 17 and 22, respectively).

When the user chooses a menu item, TaskMaster calls DoMenu, which reads the item's ID number from TaskData, strips off its "256" bit, and uses the result to call one of four subroutines, DoQuit, DoWin1, DoWin2, or DoWin3. DoQuit simply sets the QuitFlag and returns to EventLoop. The DoWin subroutines select a window (bring it to the front and make it active), then show it (make it visible if it is hidden).

The DoClose subroutine executes when the user has pressed the mouse button in the active window's close box. It simply calls HideWindow, to make the window disappear. (If it had called CloseWindow, the user wouldn't be able to redisplay the window, because it would no longer exist.)

There is also a new MenuData segment that holds the menu definitions. Note that the line for "About this program..." includes a D modifier to disable the item and that Quit's line includes a *Qq modifier, to display the OpenApple-Q key command. The rest of the program is the same as for WINDOWS, except that there's a minor change in WindowData. The window frame definition in each window parameter block now has a 1 in the second bit position, to install a close box.

Example 10-3

; MENUS displays three windows and lets the user shuffle, drag, or
 ; close them, or make them zoom or grow. It also provides three menus.
 ; The Apple menu has one item (About this program...), but it's disabled.
 ; The File menu has the Quit command and the Windows menu lets the
 ; user unhide a window and bring it to the front.

```

      absaddr on
      MCOPY Menus.macros

Menus      START
           using GlobalData

;-----
;
; Global equates used throughout the program.

ScreenMode    gequ $80           ;640 mode, no fill
MaxX          gequ 640           ;640 mode for Event Manager

           phk                   ;Set data bank to program
           plb                   ; bank to allow absolute addressing

           jsr InitStuff         ;Initialize everything
           bcs AllDone          ;Quit if initialization fails

           stz QuitFlag         ;Initialize the quit flag to 0
           jsr EventLoop        ;Let the user play

AllDone       anop               ;All is done, shut down
           PushLong Win1Ptr      ;Close the windows
           _CloseWindow
           PushLong Win2Ptr      ;Close the windows
           _CloseWindow
           PushLong Win3Ptr      ;Close the windows
           _CloseWindow

           _DeskShutDown         ;Desk Manager
           _MenuShutDown        ;Menu Manager
           _WindShutDown        ;Window Manager
           _CtlShutDown         ;Control Manager
           _EMShutDown          ;Event Manager
           _QDShutDown          ;QuickDraw II
           _MTShutDown          ;Miscellaneous Tools

           PushWord MyID         ;Discard the program's handle
           _DisposeAll

           PushWord MyID
           _MMShutDown           ;Memory Manager
           _TLShutDown          ;Tool Locator

           _Quit QuitParams      ;Do a ProDOS Quit call
           brk $F0              ;If it fails, break

END

```



```

;*****
;
; Global Data
;
;*****

GlobalData    DATA

QuitParams    dc i4'0'                ;Return to caller
               dc i'$4000'            ;Make program restartable in memory

ToolTable     dc i'NumTools'          ;No. of tool sets in table
               dc i'4,$0100'          ;QuickDraw
               dc i'5,$0100'          ;Desk Manager
               dc i'6,$0100'          ;Event Manager
               dc i'14,$0100'         ;Window Manager
               dc i'15,$0100'         ;Menu Manager
               dc i'16,$0100'         ;Control Manager

TTEnd         anop

TableSize     equ TTEnd-ToolTable-2
NumTools      equ TableSize/4

TaskRecord    anop                    ;Buffer for task record
EventWhat     ds 2
EventMessage  ds 4
EventWhen     ds 4
EventWhere    ds 4
EventModifiers ds 2
TaskData      ds 4
TaskMask      dc i4'$01FFF'          ;Make TaskMaster do all it can

Win1Ptr       ds 4                    ;Window pointers
Win2Ptr       ds 4
Win3Ptr       ds 4

MyID          ds 2                    ;This will hold the program's i.d.

VolNotFound   equ $45                 ;ProDOS error

QuitFlag      ds 2                    ;Quit flag
               END

;*****
;
; InitStuff
;
; Initializes tool sets, gets space in bank 0 for use as direct
; page by tool sets that need it, and ensures that RAM-based
; tools are in memory.
;
;*****

InitStuff     START
               using GlobalData
               using MenuData

; Initialize the Tool Locator, Memory Manager, and Miscellaneous
; Tools.

```

344 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

```

    _TLStartup                ;Tool locator
    PushWord #0               ;Memory Manager
    _MMStartup

    pla                       ;Memory Manager returns program's ID
    sta MyID

    _MTStartup                ;Misc. Tools
    ldx #3
    jsr PrepareToDie

; Get some space for the direct page we need. QuickDraw needs
; three pages and each Manager except Window needs one page.

    PushLong #0               ;Space for handle
    PushLong #$600           ;Six pages
    PushWord MyID             ;Owner
    PushWord #$C005           ;Locked, fixed, fixed bank
    PushLong #0               ;Location
    _NewHandle
    ldx #$FF
    jsr PrepareToDie

    pla                       ;Read handle and store in dp
    sta 0
    pla
    sta 2

    lda [0]                   ;Get dp location from handle
    sta 4                     ; and store at loc 4 of dp

; Initialize QuickDraw

    pha                       ;Use dp obtained from handle
    PushWord #ScreenMode     ;Mode = 640
    PushWord #160             ;Max size of scan line (in bytes)
    PushWord MyID
    _QDStartup
    ldx #4
    jsr PrepareToDie

; Initialize Event Manager

    lda 4                     ;dp to use = QD dp +$300
    clc
    adc #$300
    sta 4
    pha
    PushWord #20              ;Queue size
    PushWord #0               ;X clamp left
    PushWord #MaxX            ;X clamp right
    PushWord #0               ;Y clamp top
    PushWord #200             ;Y clamp bottom
    PushWord MyID
    _EMStartup
    ldx #6
    jsr PrepareToDie

;-----
;
; Load the RAM-based tools I need.
```

```
LoadAgain      PushLong #ToolTable
               _LoadTools
               bcc ToolsLoaded
```

```
               cmp #VolNotFound
               beq DoMount
               sec
               ldx #$FE
               jsr PrepareToDie
```

```
DoMount        anop
               jsr MountBootDisk
               cmp #1
               beq LoadAgain

               sec
               rts
```

```
; The tools have been loaded. Initialize the Window Manager, Control
; Manager, Menu Manager, and Desk Manager.
```

```
ToolsLoaded    anop
               PushWord MyID           ;Window Manager
               _WindStartup
               ldx #14
               jsr PrepareToDie

               PushLong #0             ;Prepare screen for windows
               _RefreshDesktop

               PushWord MyID           ;Control Manager
               lda 4                   ;dp to use = EM dp + $100
               clc
               adc #$100
               sta 4
               pha
               _CtlStartup
               ldx #16
               jsr PrepareToDie

               PushWord MyID           ;Menu Manager
               lda 4
               clc
               adc #$100
               pha
               _MenuStartup
               ldx #15
               jsr PrepareToDie

               _DeskStartup            ;Desk Manager
```

```
;-----
;
; Build the menu bar by inserting the three menus (right to left order).
```

```
               PushLong #0             ;Windows menu
               PushLong #WindowsMenu
               _NewMenu
               PushWord #0
               _InsertMenu
```

```

        PushLong #0                ;File menu
        PushLong #FileMenu
        _NewMenu
        PushWord #0
        _InsertMenu

        PushLong #0                ;Apple menu
        PushLong #AppleMenu
        _NewMenu
        PushWord #0
        _InsertMenu

;-----
;
; Call the Desk Manager to install the list of NDAs in the system.

        PushWord #1                ;NDAs will go in Apple menu
        _FixAppleMenu

;-----
;
; Finish off getting the menu bar ready.

        PushWord #0
        _FixMenuBar
        pla                        ;Discard menu bar height

        _DrawMenuBar              ;Display the completed bar

        jsr SetUpWindows          ;Show the windows
        _ShowCursor              ; and the mouse pointer

        jsr DrawWindows           ;Draw the window contents

        clc                      ;Clear the carry flag
        rts                      ; and return

        END

;*****
;
; Set Up Windows
;
; Opens the three windows (but does not draw their contents).
;
;*****

SetUpWindows START
        using GlobalData
        using WindowData

        PushLong #0                ;Window 1
        PushLong #Win1ParamBlock
        _NewWindow

        pla
        sta Win1Ptr
        pla

        sta Win1Ptr+2

```

```

        PushLong #0                ;Window 2
        PushLong #Win2ParamBlock
        _NewWindow

        pla
        sta Win2Ptr
        pla
        sta Win2Ptr+2

        PushLong #0                ;Window 3
        PushLong #Win3ParamBlock
        _NewWindow

        pla
        sta Win3Ptr
        pla
        sta Win3Ptr+2

        rts
        END

;*****
;
; Event Loop
;
; Display windows until user chooses "Quit".
;
;*****

EventLoop    START
              using GlobalData

Again        lda QuitFlag
              bne NoMore

              PushWord #0            ;Space for result
              PushWord #$FFFF        ;Accept any event
              PushLong #TaskRecord   ;Point to task record buffer
              _TaskMaster

              pla                    ;Is an event available?
              beq Again              ; No. Continue polling
              asl a                  ; Yes. Double it
              tax                    ; and copy it into X.

              jsr (TaskTable,x)      ;Execute the event's routine,
              bra Again              ; then resume polling

NoMore       rts

TaskTable    anop                    ;Event Manager events
              dc i'Ignore'           ; 0 null
              dc i'Ignore'           ; 1 mouse down
              dc i'Ignore'           ; 2 mouse up
              dc i'Ignore'           ; 3 key down
              dc i'Ignore'           ; 4 undefined
              dc i'Ignore'           ; 5 auto-key
              dc i'Ignore'           ; 6 update
              dc i'Ignore'           ; 7 undefined
              dc i'Ignore'           ; 8 activate
              dc i'Ignore'           ; 9 switch

```

```

dc i'Ignore'           ; 10 desk acc
dc i'Ignore'           ; 11 device driver
dc i'Ignore'           ; 12 ap
dc i'Ignore'           ; 13 ap
dc i'Ignore'           ; 14 ap
dc i'Ignore'           ; 15 ap
dc i'Ignore'           ; 16 in desktop
dc i'DoMenu'           ; 17 in system menu bar
dc i'Ignore'           ; 18 in system window
dc i'Ignore'           ; 19 in window content region
dc i'Ignore'           ; 20 in drag region (title bar)
dc i'Ignore'           ; 21 in grow box
dc i'DoClose'          ; 22 in go-away region (close box)
dc i'Ignore'           ; 23 in zoom box
dc i'Ignore'           ; 24 in information bar
dc i'Ignore'           ; 25 in right scroll bar
dc i'Ignore'           ; 26 in bottom scroll bar
dc i'Ignore'           ; 27 in frame
dc i'Ignore'           ; 28 in drop region
END

```

```

;*****
;
; Ignore
;
; This do-nothing subroutine returns to the event loop for
; events we don't want.
;
;*****

```

```

Ignore    START
          rts
          END

```

```

;*****
;
; DoMenu
;
; Called when TaskMaster tells me that a menu item has been
; selected.
;
;*****

```

```

DoMenu    START
          using GlobalData

          lda TaskData           ;Get the item ID
          and #$00FF            ; and strip off the "256" bit
          asl a                  ;Double the result
          tax                     ; and copy it to X

          jsr (ItemTable,x)      ;Call the item's subroutine,

          PushWord #0            ; then unhighlight the menu
          PushWord TaskData+2
          _HiliteMenu

          rts

```

```

ItemTable      dc i'Ignore'           ;From Apple menu
                dc i'DoQuit'          ;From File menu
                dc i'DoWin1'          ;From Windows menu
                dc i'DoWin2'
                dc i'DoWin3'

                END

;*****
;
; Do Quit
;
; Sets the quit flag.
;
;*****
DoQuit          START
                using GlobalData

                lda #1
                sta QuitFlag
                rts

                END

;*****
;
; DoWin1
;
; Selects and shows window 1 in response to menu selection.
;
;*****
DoWin1          START
                using GlobalData

                PushLong Win1Ptr
                _SelectWindow

                PushLong Win1Ptr
                _ShowWindow

                rts

                END

;*****
;
; DoWin2
;
; Selects and shows window 2 in response to menu selection.
;
;*****
DoWin2          START
                using GlobalData

                PushLong Win2Ptr
                _SelectWindow

                PushLong Win2Ptr
                _ShowWindow

                rts

                END

```

350 APPLE II GS ASSEMBLY LANGUAGE PROGRAMMING

```
;*****  
;  
; DoWin3  
;  
; Selects and shows window 3 in repsonse to menu selection.  
;  
;*****
```

```
DoWin3      START  
            using GlobalData  
  
            PushLong Win3Ptr  
            _SelectWindow  
  
            PushLong Win3Ptr  
            _ShowWindow  
  
            rts  
            END
```

```
;*****  
;  
; DoClose  
;  
; Hides the active window when user presses button in its  
; close box.  
;  
;*****
```

```
DoClose     START  
            using GlobalData  
  
            PushLong TaskData  
            _HideWindow  
  
            rts  
            END
```

```
;*****  
;  
; Menu Data  
;  
;*****
```

```
MenuData    DATA
```

```
AppleMenu   dc c'>L\N1X',i1'13'  
            dc c' LAbout this program...\N256D',i1'13' ;Item disabled  
            dc c'.'
```

```
FileMenu    dc c'>L File \N2',i1'13'  
            dc c' LQuit\N257*Qq',i1'13' ;Key command = Apple-Q  
            dc c'.'
```

```
WindowsMenu dc c'>L Windows \N3',i1'13'  
            dc c' LWindow 1\N258',i1'13'  
            dc c' LWindow 2\N259',i1'13'  
            dc c' LWindow 3\N260',i1'13'  
            dc c'.'  
            END
```



```

;*****
;
; Window Data
;
;*****

```

WindowData DATA

Win1ParamBlock anop

```

dc    i'Win1End-Win1ParamBlock'
dc    i2'%1101110110000000' Everything but info bar
dc    i4'Win1Title'            Pointer to window's title
dc    i4'0'                    Reserved (wRefCon)
dc    i2'26,0,190,620'        Zoomed content region
dc    i4'0'                    Default color table
dc    i2'0'                    Vertical origin
dc    i2'0'                    Horizontal origin
dc    i2'200'                  Data area height
dc    i2'640'                  Data area width
dc    i2'200'                  Max grow height
dc    i2'640'                  Max grow width
dc    i2'4'                    Number of pixels to scroll vertically.
dc    i2'16'                   Number of pixels to scroll

```

horizontally.

```

dc    i2'40'                   Number of pixels to page vertically.
dc    i2'160'                  Number of pixels to page horizontally.
dc    i4'0'                    Information bar text string.
dc    i2'0'                    Info bar height
dc    i4'0'                    Routine to draw shape (none, standard)
dc    i4'0'                    Routine to draw info. bar.
dc    i4'DrawWin1'              Routine to draw content.
dc    i'40,20,100,360'        Size/pos of content
dc    i4'-1'                   Window's order (-1 means topmost)
dc    i4'0'                    Window Manager allocates wind. record

```

Win1End anop

Win2ParamBlock anop

```

dc    i'Win2End-Win2ParamBlock'
dc    i2'%1101110110000000' Everything but info bar
dc    i4'Win2Title'            Pointer to window's title
dc    i4'0'                    Reserved (wRefCon)
dc    i2'26,0,190,620'        Zoomed content region
dc    i4'0'                    Default color table
dc    i2'0'                    Vertical origin
dc    i2'0'                    Horizontal origin
dc    i2'200'                  Data area height
dc    i2'640'                  Data area width
dc    i2'200'                  Max grow height
dc    i2'640'                  Max grow width
dc    i2'4'                    Number of pixels to scroll vertically.
dc    i2'16'                   Number of pixels to scroll

```

horizontally.

```

dc    i2'40'                   Number of pixels to page vertically.
dc    i2'160'                  Number of pixels to page horizontally.
dc    i4'0'                    Information bar text string.
dc    i2'0'                    Info bar height
dc    i4'0'                    Routine to draw shape (none, standard)
dc    i4'0'                    Routine to draw info. bar.
dc    i4'DrawWin2'              Routine to draw content.
dc    i'50,30,110,380'        Size/pos of content

```

```

        dc      i4'-1'                Window's order (-1 means topmost)
        dc      i4'0'                 Window Manager allocates wind. record
Win2End  anop

Win3ParamBlock anop
        dc      i'Win3End-Win3ParamBlock'
        dc      i2'%1101110110000000' Everything but info bar
        dc      i4'Win3Title'         Pointer to window's title
        dc      i4'0'                 Reserved (wRefCon)
        dc      i2'26,0,190,620'      Zoomed content region
        dc      i4'0'                 Default color table
        dc      i2'0'                 Vertical origin
        dc      i2'0'                 Horizontal origin
        dc      i2'200'                Data area height
        dc      i2'640'                Data area width
        dc      i2'200'                Max grow height
        dc      i2'640'                Max grow width
        dc      i2'4'                 Number of pixels to scroll vertically.
        dc      i2'16'                Number of pixels to scroll
horizontally.
        dc      i2'40'                 Number of pixels to page vertically.
        dc      i2'160'                Number of pixels to page horizontally.
        dc      i4'0'                 Information bar text string.
        dc      i2'0'                 Info bar height
        dc      i4'0'                 Routine to draw shape (none, standard)
        dc      i4'0'                 Routine to draw info. bar.
        dc      i4'DrawWin3'           Routine to draw content.
        dc      i'60,40,120,400'      Size/pos of content
        dc      i4'-1'                Window's order (-1 means topmost)
        dc      i4'0'                 Window Manager allocates wind. record
Win3End  anop

Win1Title      str 'Window 1'
Win2Title      str 'Window 2'
Win3Title      str 'Window 3'

PenPat1  dc h'11 11 11 11 11 11 11 11' ;Dithered blue pen pattern
        dc h'11 11 11 11 11 11 11 11'
        dc h'11 11 11 11 11 11 11 11'
        dc h'11 11 11 11 11 11 11 11'

Rect      dc i'20,20,40,40'           ;Rectangle painted in windows

END

;*****
;
; Draw Windows
;
; Fills in the content region of each window, working rearward.
;
;*****

DrawWindows  START
              using GlobalData

              PushLong Win3Ptr          ;Window 3
              _SetPort

              jsl DrawWin3

```

```

        PushLong Win3Ptr
        _ShowWindow

        PushLong Win2Ptr           ;Window 2
        _SetPort

        jsl DrawWin2

        PushLong Win2Ptr
        _ShowWindow

        PushLong Win1Ptr           ;Window 1
        _SetPort

        jsl DrawWin1

        PushLong Win1Ptr
        _ShowWindow

        rts
    END

;*****
;
; Draw Window 1
;
; Draws the contents of Window 1--a blue box on a white
; background. Called by DrawWindows initially and by the Window
; Manager when it updates the window.
;
;*****

DrawWin1 START
    using WindowData

        PushWord #3                 ;Background is white
        _SetSolidBackPat

        PushLong #PenPat1           ;Set pen pattern to blue
        _SetPenPat

        PushLong #Rect              ; and paint the rectangle
        _PaintRect

        rtl                         ;RTL required for update events
    END

;*****
;
; Draw Window 2
;
; Draws the contents of Window 2--a white box on a green
; background. Called by DrawWindows initially and by the Window
; Manager when it updates the window.
;
;*****

DrawWin2 START
    using WindowData

```

354 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

```

        PushWord #2                ;Background is green
        _SetSolidBackPat

        PushWord #3                ;Set pen pattern to white
        _SetSolidPenPat

        PushLong #Rect             ; and paint the rectangle
        _PaintRect

        rtl                        ;RTL required for update events
        END

;*****
;
; Draw Window 3
;
; Draws the contents of Window 3--a green box on a red
; background. Called by DrawWindows initially and by the Window
; Manager when it updates the window.
;
;*****

DrawWin3 START
        using WindowData

        PushWord #1
        _SetSolidBackPat

        PushWord #2                ;Set pen pattern to green
        _SetSolidPenPat

        PushLong #Rect             ; and paint the rectangle
        _PaintRect

        rtl                        ;RTL required for update events
        END

*****
*
* Prepare To Die
*
* Dies if carry is set.
*
* Error number to display is in X register.
* Assumes that Miscellaneous Tools is active.
*
*****

PrepareToDie START
        bcs RealDeath              ;Carry = 1?
        rts                        ; No. Return to caller

RealDeath      phx                  ; Yes. Goodbye, program.
                PushLong #DeathMsg
                _SysFailMgr

DeathMsg       str 'Could not handle error '

                END

```

```

;*****
;
; Mount Boot Disk
;
; Tells the user to insert the disk that contains the RAM-based
; tools.
;
;*****

MountBootDisk  START

    _Set_Prefix SetPrefixParams
    _Get_Prefix GetPrefixParams

    PushWord #0                ;Space for result
    PushWord #195              ;Column position for dialog box
    PushWord #30               ;Row position for dialog box
    PushLong #PromptStr        ;Prompt at top of dialog box
    PushLong #VolStr           ;Volume name string
    PushLong #OKStr            ;String in Button 1
    PushLong #CancelStr        ;String in Button 2
    _TLMountVolume

    pla                        ;Obtain the button number
    rts                        ; and return to caller

PromptStr    str 'Please insert the disk.'
VolStr       ds 16
OKStr        str 'OK'
CancelStr    str 'Shutdown'

GetPrefixParams dc i'7'
               dc i4'VolStr'

SetPrefixParams dc i'7'
               dc i4'BootStr'

BootStr       str '*/'

END

```

CHAPTER 11

Controls

In Chapter 9 you learned about the zoom box, scroll box, close box, and other controls you can provide in the frame of a window. By using the resources of the *Control Manager*, you can also provide controls inside a window — that is, within its content region.

This chapter is a little different from preceding chapters. I will describe the Control Manager's controls (at least the predefined ones), but I won't say much about specific tool calls, nor will I provide a programming example that uses them. Indeed, this chapter is rather short — for a very good reason. I have kept the Control Manager details to a minimum because most people won't use it, at least not directly. Instead, programmers generally access the resources of the *Dialog Manager* to do control-related operations.

When controls are involved, the Dialog Manager calls on the Control Manager behind the scenes to do the necessary work. The Dialog Manager also does some of the tasks these controls require, such as scrolling when the mouse button is pressed in a scroll bar; tasks a Control Manager-based program would have to do manually. In short, using the Dialog Manager (instead of the Control Manager) for operations that involve controls saves you from a lot of unnecessary work. It's somewhat like the benefits you gain by using the Window Manager's TaskMaster instead of the Event Manager's GetNextEvent tool.

The Apple Human Interface Guidelines are very clear as to what certain controls should do, and I will abide with the Guidelines' recommendations throughout this chapter. (Indeed, I have attempted to do so throughout this entire book.) Therefore, when I say a certain control does something or other, you can mentally follow my statement with the phrase "in accordance with the Apple Human Interface Guidelines."

Predefined Controls

The Control Manager provides four types of predefined controls. Three of them — buttons, check boxes, and radio buttons — are on/off controls (see Figure 11-1). The fourth is the familiar scroll bar.

Buttons

A *button* makes something happen the instant you click or press it with the mouse. Buttons appear as round-cornered rectangles with a title inside.

The Apple Human Interface Guidelines suggest that if you want to include buttons, you provide at least two of them: an "OK" (or "Begin", or something similar) to proceed with the operation and a "Cancel" (or perhaps "Quit") to nullify it. Note that the OK button in Figure 11-1 has a thick outline, to indicate that the user may press the Return key, instead of the mouse button, to proceed with the operation.

While you may not realize it, you have already seen Control Manager buttons in action. The Program Launcher provides four of them — Disk, Open, Close, and Quit — when you boot the Apple IIGS System Disk. It gives the Open button a thick outline, which lets you press Open to run the program whose name is highlighted.

The controls described next, check boxes and radio buttons, are used to specify options for the action the OK button will perform.

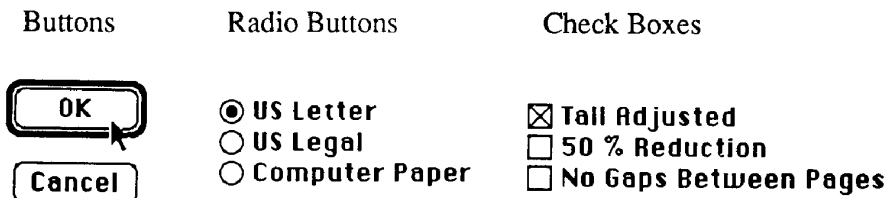


Figure 11-1

Check Boxes

A *check box* is a small square with a label to the right of it. The box contains an “X” when it has been selected (or checked); otherwise, it is blank (unchecked). Providing a group of check boxes allows the user to choose one or more options for some upcoming event.

Radio Buttons

Radio buttons are also on/off controls, but only one of them may be “on” (selected) at any given time — just like the buttons on a car radio. A radio button looks like a round check box. However, when selected, it contains a small black circle instead of an X.

Scroll Bars

The Control Manager’s scroll bars are the same as the Window Manager’s; they can have arrows, a scroll bar (or *thumb*), and gray “page” regions. However, while the scroll bars in the Window Manager’s so-called document windows only control what’s displayed in the content region, a Control Manager scroll bar can be used to control anything you want.

Scroll bars are particularly handy for letting users examine a list of data. Indeed, the System Disk’s Program Launcher provides one for just that purpose; to let you scroll through the list of available files and directories.

Scroll bars can also function as slide-type selectors, where the user can increase or decrease a program value by pressing the mouse. Slide selectors can be convenient in simulation games, for example, for changing the speed of a race car the user is “driving” or a jet plane or spaceship he or she is “flying.”

Finally, scroll bars can also serve as indicators, to show the user a particular program value in graphic form. They could, for example, be employed as the bars in a bar chart.

The Control Manager provides for a variety of controls under the broad category of *dials*. The scroll bar is the only predefined dial, but you can also (with a little more work) produce on-screen fuel gauges, thermometers, or just about any other kind of control or indicator.

Scroll Bar Components

When you add a scroll bar to a window’s frame using the Window Manager, it always comes decked out with arrows, a thumb, and (unless the content region shows the entire data area) “page” regions. However, because a

Control Manager scroll bar is a form of dial, you can equip it with as few or as many components as you want.

Figure 11-2 shows the components of scroll bars and their standard Control Manager terms. The nomenclature looks a little strange when applied to the horizontal scroll bar. While we naturally think in terms of left and right arrows, and page left and page right regions, the Control Manager thinks solely in terms of “up” and “down.” It only cares about a scroll bar’s parts, not whether the bar is vertical or horizontal.

Active and Inactive Controls

Like menu titles and items, controls may be active or inactive. An *active* control can be selected by pressing the mouse button inside it. Generally, this also highlights it (see Figure 11-3). For controls that have multiple parts, the highlighting only affects the part that the mouse pointer is in. For example, when the user presses the mouse button in an arrow of a scroll bar, only the arrow becomes highlighted.

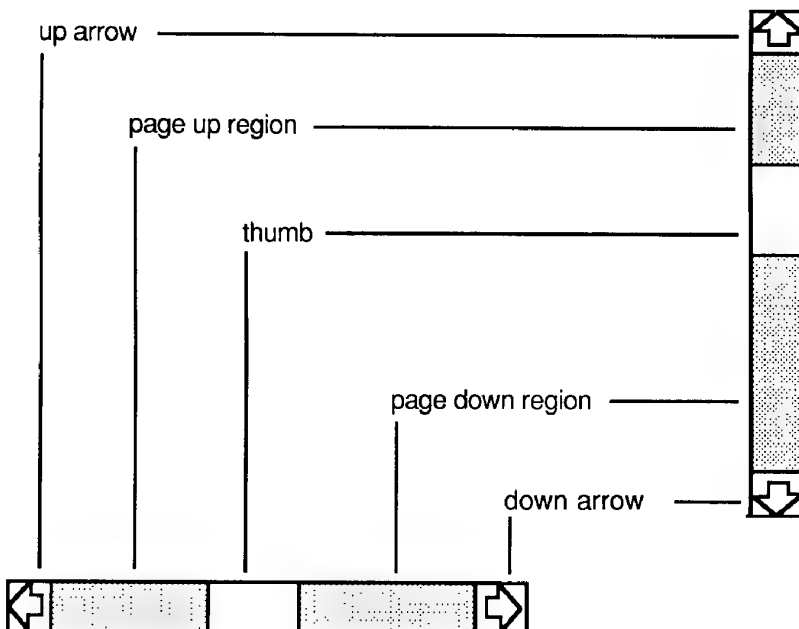
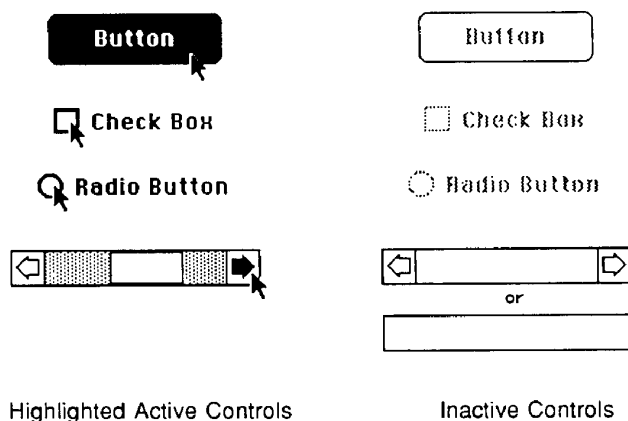


Figure 11-2

**Figure 11-3**

A control is made inactive when its operation is meaningless in the current context. For example, the IIGS Program Launcher starts with the Close button inactive, because in order to close a file, you must first open one. Inactive buttons, check boxes, and radio buttons appear dim on the screen. Inactive scroll bars look the same as active ones, but they're either lacking the thumb and paging regions or they're blank altogether (Figure 11-3 illustrates both states).

Control Manager Tool Calls

As I mentioned earlier, although the Control Manager is fully responsible for controls, you will probably find it easier to use its resources indirectly, through calls to the Dialog Manager. Thus, the only Control Manager calls that I must list are the ones that start it up and shut it down — `CtlStartup` and `CtlShutDown`; see Table 11-1. (I summarized these calls in Chapter 9, but they're worth reproducing here for completeness.)

Of course, the Control Manager also relies on other tool sets to help do its work. Before starting it, you must activate QuickDraw, the Event Manager, and the Window Manager. If your controls include text (e.g., buttons and check boxes), and you are *not* using the Dialog Manager to define them, you must also start the *Font Manager*. Refer to the Apple IIGS Toolbox Reference for details.

Table 11-1

__CtlStartUp		
Call with:	PushWord <i>ZeroPageLoc</i>	Start the Control Manager
	PushWord <i>ProgramID</i>	;Program ID
	PushWord <i>DirectPageLoc</i>	;Loc. of 1-page work area
	__CtlStartup	
Result:	None	
__CtlShutDown		
Call with:	__CtlShutDown	Shut down the Control Manager
Result:	None	

CHAPTER 12

Conducting Dialogs

It's possible to put controls and messages in the content region of a regular window, along with your picture or text, but this usually makes for a somewhat unprofessional looking program. Besides, if the window displays just a portion of an image, and lets the user scroll through the rest of it, your program would have to take care of scrolling the controls. That sounds like a lot of work — and it would be! It's much easier to create separate windows for interacting with the user and make them appear and disappear at appropriate times. The *Dialog Manager* can help you do this.

Through the services of the Dialog Manager, an application can communicate with users by displaying special-purpose windows that contain either a “dialog box” or an “alert box.” Both kinds of boxes usually contain OK and Cancel buttons, but they are very different otherwise.

A *dialog box* simply requests information from the user. For example, in a print operation, it may ask for the paper size (regular or legal) and type (single-sheet or continuous), and the name of the document to print. A dialog box is, then, a program's way of conducting a normal conversation, or dialog, with the user.

An *alert box* reports errors or issues warnings. While a dialog box generally appears as the result of something the user has done (e.g., choosing

“Print” in a menu), an alert box appears only when something has gone wrong or some unusual situation must be brought to the user’s attention.

Dialog Boxes

Figure 12-1 shows a typical dialog box, one in which the user presses buttons, checks boxes, and fills in blanks. Besides these items, a dialog box may contain controls (such as a close box, zoom box, or scroll bars), graphics images (icons or QuickDraw pictures), or anything else the application wants to put in it. The Dialog Manager provides for two kinds of dialog boxes, “modal” and “modeless.”

Modal Dialog Boxes

A modal dialog box is one that requires the user to respond before doing anything else. Once it appears, only pressing its OK or Cancel button makes it disappear; clicking the mouse button anywhere else just makes the speaker beep. In other words, the user is locked into the state or “mode” of having to respond to whatever the dialog box wants. In this respect, a modal dialog box is similar to an alert box. Note that the dialog box shown in Figure 12-1 is of the modal variety.

You can also set up modal dialog boxes that have no buttons at all. This is handy for displaying a “Please wait” type message while the application is performing a lengthy operation, such as sorting data, printing a document, or loading a file from disk.

Modeless Dialog Boxes

Unlike a modal dialog box, a modeless dialog box requires no attention whatsoever from the user. He or she can work in a document (i.e., Window Manager) window, select from menus, or do anything else the application allows, as if the dialog box isn’t even there! A modeless dialog box is, then, something the application provides strictly for the user’s convenience; he or she can use it or not.

A modeless title box can also take on the attributes of a regular document window. That is, an application can (and, in most cases, should) set up one that can be moved, made inactive or active, or closed like a document window. Figure 12-2 shows a modeless dialog box that a word-processing program might produce. Here, you could make the box disappear by clicking in its close box or, if it’s the active window, choosing Close from the file menu.

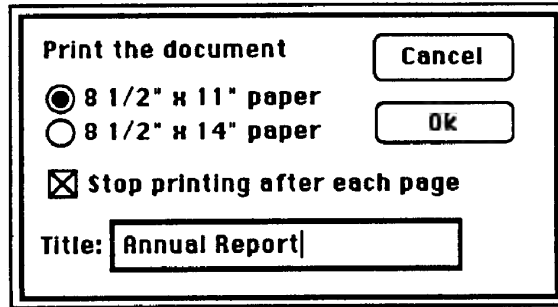


Figure 12-1

A modeless dialog box doesn't necessarily disappear when the user presses one of its buttons, however, as a modal box does. The application often keeps it around for future use. The Change box in Figure 12-2 is one that should be retained after the user presses a button; he or she may want to make more changes later, or maybe even right away.

Alert Boxes

As I mentioned at the beginning of this chapter, an alert box reports errors or issues warnings to the user when something significant has happened (or is about to happen) or something has gone wrong. Figure 12-3 shows an

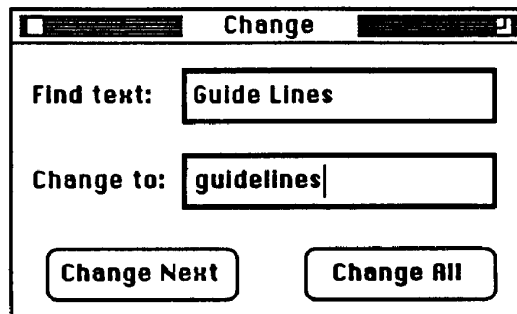


Figure 12-2

alert box that might appear when a user has selected a Quit command without first saving the active document.

Note that the *Don't Erase* button is the default here (it has a thick outline) — as it indeed should be! The Quit operation will be canceled if the user presses the Return key. Because alert boxes generally signal some drastic action, the button that lets the user back out of the operation should always be the default. It should choose “Don’t” rather than “Do”.

Types of Alert Boxes

There are three types of predefined alert boxes — Stop, Note, and Caution — each indicated by a unique icon at the top left-hand corner. Figure 12-3 shows the Caution icon. The icons for Stop and Note alerts are similar, but instead of an exclamation point, they show a hand and a “talking” face, respectively.

According to the *Apple Human Interface Guidelines*:

- A Stop alert box should signal a serious problem that requires remedial action by the user. One might appear when a disk is full or the user has removed a disk from a drive.
- A Note alert box should signal a minor mistake that wouldn’t produce disastrous consequences if left as is.
- A Caution alert box should indicate an operation that may or may not have undesirable results if it’s allowed to continue.

The Dialog Manager provides the tool calls `StopAlert`, `NoteAlert`, and `CautionAlert` to create these alert box types. Except for the fact that each call displays a different icon at the top left corner, they are identical. There is also a more general-purpose call, `Alert`, that doesn’t put any icon in the box; you can specify one of your own or omit it entirely.

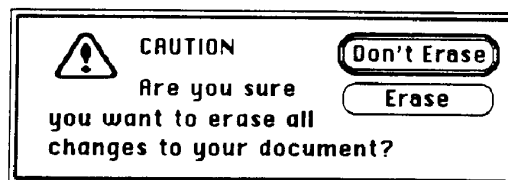


Figure 12-3

Stages of an Alert

If the user makes some minor mistake, such as trying to print a page whose number doesn't exist, you may want to start an alert sequence without displaying an alert box.

The Dialog Manager keeps track of four successive *stages* of an alert, and lets your program define a response for each one. That way, if a user persists in making the same mistake, the program can issue increasingly helpful (or, like a bill collector, increasingly sterner) messages. For example, you could make the first two occurrences of a mistake produce a beep and the next two bring up an alert box. This would be reasonable for a mistake that was probably accidental, such as choosing Cut when no block of text has been selected.

Programming Dialogs and Alerts

To include dialog or alert boxes in your program, begin by starting the Memory Manager, Desk Manager, QuickDraw, the Event Manager, Window Manager, Control Manager, and the Line Editor, in that order. The Line Editor, or LineEdit for short, requires a page of working space in bank 0. Allocate that space in your NewHandle call and start LineEdit with a sequence of the form:

```

PushWord MyID      ; Program ID from Memory Manager
lda 4              ; ZP to use = QD ZP + $500
clc
adc #$100
sta 4
pha
LEStartup
ldx #20
jsr PrepareToDie

```

That done, start the Dialog Manager with a DialogStartup call (see Table 12-1). Then, if your program uses menus, start the Menu Manager.

What the rest of the program contains depends on your application. If it includes dialog or alert boxes, you will have to create them and specify which items (buttons, scroll bars, text, etc.) belong in them.

Unlike windows and menus, however, dialog and alert boxes — or

Table 12-1

__DialogStartup		Start the Dialog Manager
Call with:	PushWord <i>ProgramID</i> ;Program ID	
	__DialogStartup	
Result:	None	
Note:	The Dialog Manager uses the Control Manager's direct page for working space.	
__DialogShutDown		Shut down the Dialog Manager
Call with:	__DialogShutDown	
Result:	None	
__LEStartup		Start the Line Editor
Call with:	PushWord <i>ProgramID</i> ;Program ID	
	PushWord <i>DirectPageLoc</i> ;Loc. of 1-page work area	
	__LEStartup	
Result:	None	
__LEShutDown		Shut down the Line Editor
Call with:	__LEShutDown	
Result:	None	

rather, the *windows* that hold them — are not necessarily created when the program starts. (You certainly shouldn't begin by confronting the user with an alert, for instance.) Instead, you can create these windows when they're needed. You may, for example, create a dialog window when the user chooses a menu item.

Once a dialog or alert window is active, your program must monitor the user's activities, to see what he or she has selected. Finally, you must remove a window from the screen when it's no longer needed.

The Dialog Manager provides tool calls for all these activities, and I will describe them shortly. But first I must discuss *item lists*, which are required to create both dialogs and alerts.

Item Lists

An item list is a summary of the specifications for a particular item in a dialog or alert box. It tells the Dialog Manager everything necessary to make that item an integral part of the box. An item list contains the following parameters:

- An *ID number* that identifies the item in the dialog. All subsequent tool calls that involve the item will refer to it by this number.
- A *display rectangle* that specifies the location of the item, in the local coordinates of the box.
- The *item type*; that is, button, check box, radio button, text, user-defined control, or whatever.
- An *item descriptor*. This is generally a pointer to a text string, such as the label in a button, the title beside a check box or radio button, or a message to the user.
- An *item value*; an initial value for the item.
- A *flag* that tells whether the item should be visible or invisible and, for some items, gives specific information (the group or “family” number of a radio button, whether a scroll bar is horizontal or vertical).
- The *color table* QuickDraw should use to draw the item.

Let’s take the parameters one at a time.

ID Number

The ID number parameter can range from 1 to 255, which means that you can put up to 255 items in a single dialog or alert box. The item numbered 1 becomes the *default*; the control that’s activated if the user presses Return. The Dialog Manager assumes item 1 is a button and gives it a bold outline. (After all, assigning 1 to anything except a button is meaningless.)

In general, item 1 should be the OK button and item 2 the Cancel button — or vice versa, in an alert box. Of course, if you don’t want a default button, don’t number any item 1.

Display Rectangle

The display rectangle, `ItemRect`, is an imaginary box in which the item is to be drawn. It specifies the location of the item, in the local coordinates of the box.

Item Type

The Dialog Manager provides for 12 different types of items, as summarized in Table 12-2.

Table 12-2

Type	Value	Description
ButtonItem	10	Simple button.
CheckItem	11	Check box.
RadioItem	12	Radio button.
ScrollBarItem	13	Special scroll bar for dialogs.
UserCtlItem	14	User-defined control.
StatText	15	Static text (up to 255 characters); text that cannot be edited.
LongStatText	16	Long static text (up to 32,767 characters).
EditLine	17	<i>Dialogs only.</i> A line of text that can be edited.
IconItem	18	<i>Dialogs only.</i> Icon.
PicItem	19	QuickDraw picture.
UserItem	20	<i>Dialogs only.</i> A user-defined item, such as a string that changes each time the box appears.
UserCtlItem2	21	Another user-defined control.

The Dialog Manager recognizes these types by numbers between 10 and 21, so you normally set up a list of the types your program uses with a series of Equates (equ's). A program that uses all 12 types would contain:

```

; Item types for Dialog Manager.

ButtonItem      equ 10    ;Button
CheckItem       equ 11    ;Check box
RadioItem       equ 12    ;Radio button
ScrollBarItem   equ 13    ;Scroll bar
UserCtlItem     equ 14    ;User-defined control
StatText        equ 15    ;Static text
LongStatText    equ 16    ;Long static text
EditLine        equ 17    ;Editable text
IconItem        equ 18    ;Icon
PicItem         equ 19    ;QuickDraw picture
UserItem        equ 20    ;User-defined item
UserCtlItem2    equ 21    ;Another user-defined control

```

In addition, if you add \$8000 to the value of an item type, the Dialog Manager will *disable* it; that is, make it unselectable. Text should be disabled (of course), but you may also want to disable a user-defined item, or an icon or QuickDraw picture. You may even want to disable a control, so the user can't do anything with it!

Unlike the Control Manager, the Dialog Manager does not provide a way to make a control inactive (i.e., show it dimmed). You can only disable it, and a disabled control looks just like an enabled one.

By convention, you would set up the \$8000 value as, perhaps,

```
ItemDisable equ $8000
```

and disable an item by specifying its type with the form:

```
EditLine+ItemDisable
```

Figure 12-4 shows an example of the various item types, with some of them disabled.

Having mentioned the EditLine item, I should mention the commands you can use to edit one. They are:

- Clicking in an item displays a blinking vertical line, to show the place where text may be inserted.

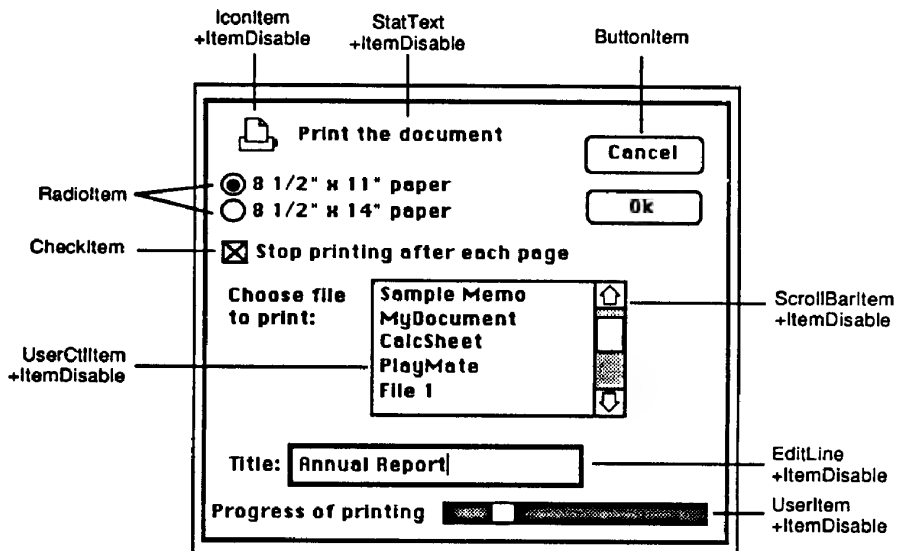


Figure 12-4

- Pressing the left or right arrow key moves the insertion point by 1 character. Pressing an arrow key with OpenApple down moves the insertion point to the beginning or end of the line. Pressing one with Option down moves the insertion point to the preceding or following word.
- Dragging over text selects it for replacement by whatever the user types. Double-clicking selects a word, while triple-clicking selects the entire line. Double-clicking followed by dragging extends or shortens the selection a word at a time.
- Pressing Delete deletes the current selection or the character preceding the insertion point.
- Control-F deletes the character that follows the insertion point, or the current selection.
- Control-Y deletes from the insertion point to the end of the line, or the current selection.
- Control-X deletes the entire line, or the current selection.
- OpenApple-X, -C, and -V do the same Cut, Copy, and Paste functions as they do in the regular editor.

Item Descriptor and Value

For a Button, Check, Radio, or StatText item, the item descriptor is a pointer to a text string. This string must be in standard ProDOS format, so you normally use the *str* macro to define it. For example,

```
OKButton str 'OK'
```

The item value for a Button or StatText item is zero, but for a Check or Radio item, make it either 0 (off) or 1 (on).

For a LongStatText (long static text) item, the item descriptor must point to the beginning of the string and the item value must contain the string's length in bytes. The following string uses ASCII carriage return (\$0D) characters to break the the text between lines:

```
VeryLongText  dc c'this text is really very
                long. ',h'0D'
                dc c'It just goes on . . . ',h'0D'
                dc c'and on . . . ',h'0D'
```

```

        dc  c'and on . . . ',h'OD'
        dc  c'and on . . . ',h'OD'
        . .
        . .
        dc  c'That's the beauty of using
            LongStatText:',h'OD'
        dc  c'You can produce many more',h'OD'
        dc  c'characters than with regular
            StatText.'
EndLongText    anop

```

The *anop* at the end lets you define the item value as

```
EndLongText-VeryLongText
```

For an *EditLine* item, the item descriptor points to a string that is to appear when the dialog window comes up on the screen and the item value is the maximum allowed length of the string. For example, if the user is to enter a ProDOS filename, you could limit the string to 15 characters and define the default text with, perhaps,

```
EditLString    str    'Untitled'
```

To omit the default text, set the item descriptor to 0.

Item Flag

The item flag is a word-size value that is only meaningful for buttons (both simple and radio) and scroll bars. Set it to 0 for any other item type.

For a simple button, the flag can only have two values: 0 to draw a round-cornered button or 2 to draw one with square corners. For a radio button, the flag tells whether the button is visible and assigns it to a “family” (see Figure 12-5). Since turning on one radio button turns off all others in a family, the flag lets the Dialog Manager know which buttons are related. A scroll bar’s item flag (also shown in Figure 12-5) assigns its controls and specifies whether the bar is horizontal or vertical.

Tool Calls for Dialog Boxes

Table 12-3 shows the most useful tool calls for dialog boxes. In general, you would call them in the order they’re listed; that is, you would call

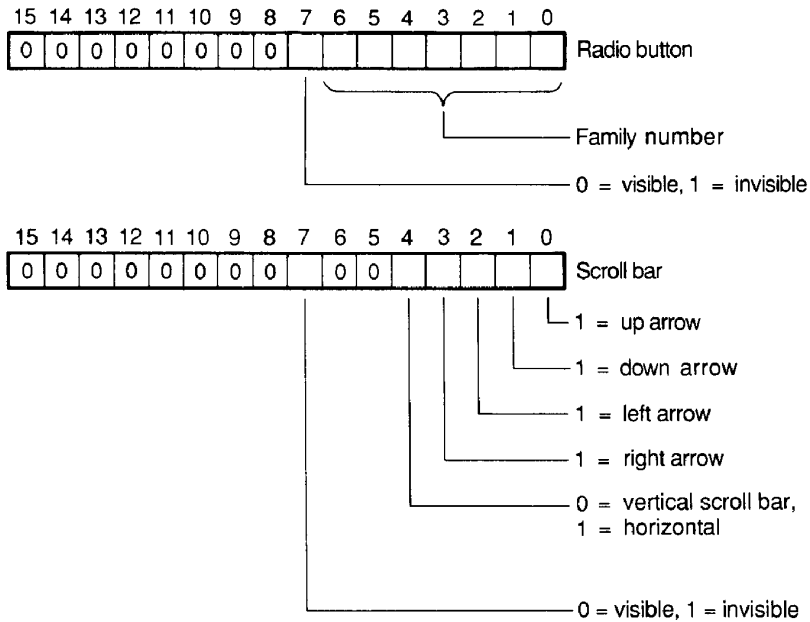


Figure 12-5

NewModalDialog or NewModeLessDialog to create any dialogs you need, then call NewItem for each item you want to add to the dialog. Finally, when you no longer need a dialog, you would call CloseDialog to get rid of it.

Table 12-3

Creating and Disposing of Dialogs		
<u>NewModalDialog</u>		Create a new modal dialog box
Call with:	PushLong #0	;Space for result
	PushLong <i>dBoundsRect</i>	;Pointer to window rectangle
	PushLong <i>dVisible</i>	;1 if visible, 0 if not
	PushLong <i>dRefCon</i>	;Any value you'd like to associate with this
		; window
<u>NewModalDialog</u>		
Result:	Pointer to dialog's port (long word)	
Note:	The top coordinate should be at least 25 points below the top of the screen, to allow for the menu bar.	

Table 12-3 (cont.)

Creating and Disposing of Dialogs (cont.)		
<hr/>		
__NewModelessDialog	Create a new modeless dialog box	
Call with:	PushLong #0	;Space for result
	PushLong <i>dBoundsRect</i>	;Pointer to window rectangle
	PushLong <i>dTitle</i>	;Pointer to title string (0 for no title)
	PushLong <i>dBehind</i>	;Pointer to window the dialog should be behind; ; - 1 puts it in front
	PushWord <i>dFlag</i>	;Flags describing the window's frame (see ; Figure 9-5)
	PushLong <i>dRefCon</i>	;Any value you'd like to associate with this ; window
	PushLong <i>FullSize</i>	;Pointer to Rect defining window's zoomed size
	__NewModelessDialog	
Result:	Pointer to dialog's port (long word)	
__CloseDialog	Close a dialog window	
Call with:	PushLong <i>theDialog</i>	;Pointer to dialog's port
	__CloseDialog	
Result:	None	
Note:	This is simply the Dialog Manager's equivalent of the Window Manager's CloseWindow call.	

Creating and Removing Items		
<hr/>		
__NewItem	Add a new item to the dialog's item list	
Call with:	PushLong <i>theDialog</i>	;Pointer to dialog's port
	PushWord <i>ItemID</i>	;ID number (1 to 255)
	PushLong <i>ItemRect</i>	;Pointer to display rectangle
	PushWord <i>ItemType</i>	;See table 12-2
	PushLong <i>ItemDescr</i>	;Item descriptor (e.g., string pointer)
	PushWord <i>ItemValue</i>	;Item value
	PushWord <i>ItemFlag</i>	;See figure 12-5 and related text
	PushLong <i>ItemColor</i>	;Pointer to item's color table (0 for default)
	__NewItem	
Result:	None	
Note:	See "Item Lists" for a description of these inputs.	
__RemoveItem	Remove an item from the dialog	
Call with:	PushLong <i>theDialog</i>	;Pointer to dialog's port
	PushWord <i>ItemID</i>	;ID number of item
	__RemoveItem	
Result:	None	

Handling Dialog Events		
<hr/>		
__Modal Dialog	Handle modal events	
Call with:	PushWord #0	;Space for result
	PushLong <i>filterProc</i>	;Pointer to a filter procedure
	__ModalDialog	
Result:	ID number of item hit (word).	

Table 12-3 (cont.)

Handling Dialog Events (cont.)		
<hr/>		
<code>__IsDialogEvent</code>		Check for modeless events
Call with:	<code>PushWord #0</code>	;Space for result
	<code>PushLong <i>theEvent</i></code>	;Pointer to the event record
	<code>__IsDialogEvent</code>	
Result:	A word. Nonzero if <i>theEvent</i> is a dialog event	
<hr/>		
<code>__DialogSelect</code>		Handle modeless event
Call with:	<code>PushWord #0</code>	;Space for result
	<code>PushLong <i>theEvent</i></code>	;Pointer to the event record
	<code>PushLong <i>theDialog</i></code>	;Pointer to variable at which to store the dialog
		; pointer
	<code>PushLong <i>itemHit</i></code>	;Pointer to an integer at which to store the
		; item number
	<code>__DialogSelect</code>	
Result:	A word. Nonzero if the event involves an enabled dialog item.	

Get Text		
<hr/>		
<code>__GetText</code>		Get text from dialog box
Call with:	<code>PushLong <i>the Dialog</i></code>	;Pointer to the dialog
	<code>PushLong <i>itemID</i></code>	;ID number of item
	<code>PushLong <i>theString</i></code>	;Pointer to a buffer to put the text in
	<code>__GetText</code>	
Result:	None	
Note:	Space for the string must be allocated before calling <code>GetText</code> .	

Handling Modal Events

To handle events in a modal dialog, simply call `ModalDialog` immediately after displaying the dialog box. If the front window is a modal dialog, `ModalDialog` monitors it and handles any events that occur. If an event is an enabled dialog item, it returns with the item's ID on the stack. If you dialog has only one enabled item (say, only an OK button), you can discard the ID and close the window with `CloseDialog`. Otherwise, you must use the ID to determine which item caused the "hit."

Note that `ModalDialog` takes a single input: a pointer to a "filter" procedure. If you push a 0 onto the stack, `ModalDialog` uses a default filter procedure in which it returns the ID of the default button (1) if the user presses Return. The *Apple IIGS Toolbox Reference* has full details on designing your own filter procedures, in case you're interested.

Handling Modeless Events

Handling events in a modeless dialog is a little trickier, in that it involves two calls: `IsDialogEvent` to check for modeless events and `DialogSelect` to actually handle the event. Both calls belong in your event loop, immediately following the `TaskMaster` calls.

`IsDialogEvent` takes one input: a pointer to the same event record the `TaskMaster` uses. The word it returns on the stack has a nonzero value (for TRUE) if the event has occurred in a dialog window; otherwise, the word contains zero (for FALSE). A nonzero result is your signal to pass the event to `DialogSelect`.

While `IsDialogEvent` simply indicates whether a dialog event has occurred, `DialogSelect` tells your program to the event record (same as for `IsDialogEvent`), a pointer to a 2-word memory location that will hold the dialog pointer, and a pointer to a 1-word location that will hold the item number. `DialogSelect` returns the same kind of TRUE/FALSE indicator as `IsDialogEvent`. Here, TRUE indicates that the event involves an enabled item.

Like the Window Manager's `TaskMaster`, `DialogSelect` can take care of some events itself. Specifically:

- If the event is an activate or update for a dialog window, `DialogSelect` activates or updates the window and returns FALSE.
- If the event is a key-down or auto-key event and an `EditLine` item is enabled, `DialogSelect` returns TRUE. In the absence of an enabled `EditLine`, key-down and auto-key events generate a FALSE result.
- If the mouse button is released inside an enabled control, `DialogSelect` returns TRUE; otherwise, it returns FALSE.
- If the mouse button is pressed in any other enabled item (e.g., an icon), `DialogSelect` returns TRUE. For mouse-down events in any disabled item or in no item, `DialogSelect` returns FALSE.

In short, then, your program must respond to TRUE results and ignore FALSE ones. Example 12-1 shows the kind of code needed to deal with modeless dialog events. To keep things simple, I'm assuming the application has only one modeless dialog window.

Get Text

The final tool call, `GetIText`, is a handy one. It lets you read the text of an `EditLine` in a dialog box. After all, if you're allowing the user to enter

something, your program must have a way of *reading* what he or she has typed.

Example 12-1

```

Again      anop

           lda QuitFlag      ;Quit flag still zero?
           bne AllDone      ; No. Exit

           PushWord #0
           PushWord #-1
           PushLong #EventRecord
           _TaskMaster

           pla               ;Has an event occurred?
           beq CheckDM      ; Maybe so. Check with the Dialog Mgr.

           asl a             ; Yes. Handle it
           tax
           jsr (TaskTable,x)
           bra Again

CheckDM     PushWord #0
           PushLong #EventRecord
           _IsDialogEvent

           pla               ;Dialog event?
           beq Again        ; No. Continue polling

           PushWord #0      ; Yes. Read it
           PushLong #EventRecord
           PushLong #theDialog
           PushLong #ItemHit
           _DialogSelect

           pla               ;Did DialogSelect deal with it?
           beq Again        ; Yes. Continue polling

           lda ItemHit      ; No. Application must handle it
           asl a
           tax

           jsr (DEventTable,x)

           bra Again
           rts

theDialog  ds 4             ;Space for dialog pointer
ItemHit    ds 2             ;Space for item number

; Dialog event table

DEventTable dc i'Ignore'    ;There's no item 0
            dc i'DoItem1'   ;Item 1
            dc i'DoItem2'   ;Item 2
            (etc.)

```

Example Dialog Box Program

Example 12-2 shows a program called MODALD that displays a modal dialog box when the user selects "About this program . . ." from an Apple menu on the menu bar. Note that MODALD is simply an enhanced version of MENUS with code added for the menu and dialog box.

Example 12-2

```
; MODALD is an enhanced version of MENUS in which choosing "About this
; program..." from the Apple menu brings on a modal dialog box that shows
; "Copyright 1987 by Leo Scanlon" and an OK button. If the user chooses
; this item again, the box shows "As I said before:" in addition to the
; original text.

        absaddr on
        MCOPY Modald.macros

Modald      START
            using GlobalData

; -----
;
; Global equates used throughout the program.

ScreenMode    gequ $80                ;640 mode, no fill
MaxX          gequ 640                ;640 mode for Event Manager

            phk                        ;Set data bank to program
            plb                        ; bank to allow absolute addressing

            jsr InitStuff              ;Initialize everything
            bcs AllDone               ;Quit if initialization fails

            stz QuitFlag              ;Initialize the quit flag to 0
            jsr EventLoop             ;Let the user play

AllDone      anop                    ;All is done, shut down
            PushLong Win1Ptr          ;Close the windows
            _CloseWindow
            PushLong Win2Ptr
            _CloseWindow
            PushLong Win3Ptr
            _CloseWindow

            _DeskShutDown             ;Desk Manager
            _MenuShutDown             ;Menu Manager
            _WindShutDown             ;Window Manager
            _DialogShutDown           ;Dialog Manager
            _CtlShutDown              ;Control Manager
            _EMShutDown               ;Event Manager
            _LEShutDown               ;LineEdit
            _QDShutDown               ;QuickDraw II
            _MTShutDown               ;Miscellaneous Tools
```


380 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

```
; Item types for dialog box.
```

```
ButtonItem    equ 10
StatText      equ 15
```

```
; To disable an item type, add ItemDisable to it.
```

```
ItemDisable   equ $8000
```

```
END
```

```
;*****
```

```
;
```

```
; InitStuff
```

```
;
```

```
; Initializes tool sets, gets space in bank 0 for use as direct  
; page by tool sets that need it, and ensures that RAM-based  
; tools are in memory.
```

```
;
```

```
;*****
```

```
InitStuff      START
                using GlobalData
                using MenuData
```

```
; Initialize the Tool Locator, Memory Manager, and Miscellaneous  
; Tools.
```

```
    _TLStartup          ;Tool locator

    PushWord #0          ;Memory Manager
    _MMStartup

    pla                 ;Memory Manager returns program's ID
    sta MyID

    _MTStartup          ;Misc. Tools
    ldx #3
    jsr PrepareToDie
```

```
; Get some space for the direct page we need. QuickDraw needs  
; three pages and each Manager except Window needs one page.
```

```
    PushLong #0         ;Space for handle
    PushLong #$700      ;Seven pages
    PushWord MyID       ;Owner
    PushWord #$C001     ;Locked, fixed, fixed bank
    PushLong #0         ;Location
    _NewHandle
    ldx #$FF
    jsr PrepareToDie

    pla                 ;Read handle and store in dp
    sta 0
    pla
    sta 2
```

```

        lda [0]                ;Get dp location from handle
        sta 4                  ; and store at loc 4 of dp

; Initialize QuickDraw

        pha                    ;Use dp obtained from handle
        PushWord #ScreenMode   ;Mode = 640
        PushWord #160          ;Max size of scan line (in bytes)
        PushWord MyID
        _QDStartup
        ldx #4
        jsr PrepareToDie

; Initialize Event Manager

        lda 4                  ;dp to use = QD dp + $300
        clc
        adc #$300
        sta 4
        pha
        PushWord #20           ;Queue size
        PushWord #0            ;X clamp left
        PushWord #MaxX         ;X clamp right
        PushWord #0            ;Y clamp top
        PushWord #200          ;Y clamp bottom
        PushWord MyID
        _EMStartup
        ldx #6
        jsr PrepareToDie

;-----
;
; Load the RAM-based tools I need.

LoadAgain    PushLong #ToolTable
              _LoadTools
              bcc ToolsLoaded

              cmp #VolNotFound
              beq DoMount
              sec
              ldx #$FE
              jsr PrepareToDie

DoMount      anop
              jsr MountBootDisk
              cmp #1
              beq LoadAgain

              sec
              rts

; The tools have been loaded. Initialize the Window Manager, Control
; Manager, LineEdit, Dialog Manager, Menu Manager, and Desk Manager.

ToolsLoaded  anop
              PushWord MyID          ;Window Manager
              _WindStartup
              ldx #14
              jsr PrepareToDie

```

```

        PushLong #0                ;Prepare screen for windows
        _RefreshDesktop

        PushWord MyID              ;Control Manager
        lda 4                      ;dp to use = EM dp + $100
        clc
        adc #$100
        sta 4
        pha
        _CtlStartup
        ldx #16
        jsr PrepareToDie

        PushWord MyID              ;LineEdit
        lda 4
        clc
        adc #$100
        sta 4
        pha
        _LEStartup
        ldx #20
        jsr PrepareToDie

        PushWord MyID              ;Dialog Manager
        _DialogStartup
        ldx #21
        jsr PrepareToDie

        PushWord MyID              ;Menu Manager
        lda 4
        clc
        adc #$100
        pha
        _MenuStartup
        ldx #15
        jsr PrepareToDie

        _DeskStartup               ;Desk Manager

;-----
;
; Build the menu bar by inserting the three menus (right to left order).

        PushLong #0                ;Windows menu
        PushLong #WindowsMenu
        _NewMenu
        PushWord #0
        _InsertMenu

        PushLong #0                ;File menu
        PushLong #FileMenu
        _NewMenu
        PushWord #0
        _InsertMenu

        PushLong #0                ;Apple menu
        PushLong #AppleMenu
        _NewMenu
        PushWord #0
        _InsertMenu

```



```

;-----
;
; Call the Desk Manager to install the list of NDAs in the system.

        PushWord #1                ;NDAs will go in Apple menu
        _FixAppleMenu

;-----
;
; Finish off getting the menu bar ready.

        PushWord #0
        _FixMenuBar
        pla                        ;Discard menu bar height

        _DrawMenuBar                ;Display the completed bar

        jsr SetUpWindows            ;Show the windows
        _ShowCursor                ; and the mouse pointer

        jsr DrawWindows             ;Draw the window contents

        clc                        ;Clear the carry flag
        rts                        ; and return

        END

;*****
;
; Set Up Windows
;
; Opens the three windows (but does not draw their contents).
;
;*****

SetUpWindows START
        using GlobalData
        using WindowData

        PushLong #0                ;Window 1
        PushLong #Win1ParamBlock
        _NewWindow

        pla
        sta Win1Ptr
        pla
        sta Win1Ptr+2

        PushLong #0                ;Window 2
        PushLong #Win2ParamBlock
        _NewWindow

        pla
        sta Win2Ptr
        pla
        sta Win2Ptr+2

        PushLong #0                ;Window 3
        PushLong #Win3ParamBlock
        _NewWindow

```

384 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

```

        pla
        sta Win3Ptr
        pla
        sta Win3Ptr+2

        rts
        END

;*****
;
; Event Loop
;
; Display windows until user chooses "Quit".
;
;*****

EventLoop      START
                using GlobalData

Again          lda QuitFlag
                bne NoMore

                PushWord #0                ;Space for result
                PushWord #$FFFF           ;Accept any event
                PushLong #TaskRecord      ;Point to task record buffer
                _TaskMaster

                pla                        ;Is an event available?
                beq Again                 ; No. Continue polling
                asl a                      ; Yes. Double it
                tax                        ; and copy it into X.

                jsr (TaskTable,x)         ;Execute the event's routine,
                bra Again                 ; then resume polling

NoMore         rts

TaskTable      anop                      ;Event Manager events
                dc i'Ignore'             ; 0 null
                dc i'Ignore'             ; 1 mouse down
                dc i'Ignore'             ; 2 mouse up
                dc i'Ignore'             ; 3 key down
                dc i'Ignore'             ; 4 undefined
                dc i'Ignore'             ; 5 auto-key
                dc i'Ignore'             ; 6 update
                dc i'Ignore'             ; 7 undefined
                dc i'Ignore'             ; 8 activate
                dc i'Ignore'             ; 9 switch
                dc i'Ignore'             ; 10 desk acc
                dc i'Ignore'             ; 11 device driver
                dc i'Ignore'             ; 12 ap
                dc i'Ignore'             ; 13 ap
                dc i'Ignore'             ; 14 ap
                dc i'Ignore'             ; 15 ap
                dc i'Ignore'             ; 16 in desktop
                dc i'DoMenu'             ; 17 in system menu bar
                dc i'Ignore'             ; 18 in system window
                dc i'Ignore'             ; 19 in window content region
                dc i'Ignore'             ; 20 in drag region (title bar)
                dc i'Ignore'             ; 21 in grow box

```

```

        dc i'DoClose'          ; 22 in go-away region (close box)
        dc i'Ignore'           ; 23 in zoom box
        dc i'Ignore'           ; 24 in information bar
        dc i'Ignore'           ; 25 in right scroll bar
        dc i'Ignore'           ; 26 in bottom scroll bar
        dc i'Ignore'           ; 27 in frame
        dc i'Ignore'           ; 28 in drop region
    END

;*****
;
; Ignore
;
; This do-nothing subroutine returns to the event loop for
; events we don't want.
;
;*****

Ignore    START
          rts
          END

;*****
;
; DoMenu
;
; Called when TaskMaster tells me that a menu item has been
; selected.
;
;*****

DoMenu    START
          using GlobalData

          lda TaskData          ;Get the item ID
          and #$00FF            ; and strip off the "256" bit
          asl a                 ;Double the result
          tax                   ; and copy it to X

          jsr (ItemTable,x)      ;Call the item's subroutine,

          PushWord #0            ; then unhighlight the menu
          PushWord TaskData+2
          _HiliteMenu

          rts

ItemTable dc i'DoAbout'         ;From Apple menu
          dc i'DoQuit'          ;From File menu
          dc i'DoWin1'          ;From Windows menu
          dc i'DoWin2'
          dc i'DoWin3'

          END

;*****
;
; Do Quit
;
; Sets the quit flag.
;
;*****

```

386 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

```

DoQuit          START
                using GlobalData

                lda #1
                sta QuitFlag
                rts

                END

;*****
;
; Do About
;
; Brings up About box and waits until user presses OK button
; before putting it away.
;
;*****

DoAbout         START
                using GlobalData
                using WindowData

                PushLong #0                ;Space for result (pointer)
                PushLong #DRect            ;Pointer to content rect.
                PushWord #1                ;Make window visible
                PushLong #0                ;No particular value
                _NewModalDialog

                pla                        ;Store pointer to port
                sta MDialogPtr
                pla
                sta MDialogPtr+2

                PushLong MDialogPtr        ;Set up the OK button
                PushWord #1                ;Item #1 (default)
                PushLong #OKBtnRect        ;Display rectangle
                PushWord #ButtonItem        ;OK is a simple button
                PushLong #ButtonText       ;Pointer to its text
                PushWord #0                ;Initial value
                PushWord #0                ;Default flag
                PushLong #0                ;Default color table
                _NewItem

                lda NotFirstTime           ;First time for this box?
                beq ShowText2

                PushLong MDialogPtr        ; No. Display "As I said before:"
                PushWord #2                ;Item #2
                PushLong #TextRect1
                PushWord #StatText+ItemDisable
                PushLong #Text1
                PushWord #0
                PushWord #0
                PushLong #0
                _NewItem

ShowText2       PushLong MDialogPtr        ;Display copyright message
                PushWord #3                ;Item #3
                PushLong #TextRect2
                PushWord #StatText+ItemDisable
                PushLong #Text2
                PushWord #0

```

```

        PushWord #0
        PushLong #0
        _NewItem

        PushWord #0           ;Box only has one selectable item
        PushLong #0
        _ModalDialog

        pla                   ;I know which item it was. Ignore it.

        PushLong MDialogPtr   ;Take the box off the screen
        _CloseDialog

        lda #1                ;From here on, display message 1
        sta NotFirstTime
        rts

    END

;*****
;
; DoWin1
;
; Selects and shows window 1 in response to menu selection.
;
;*****

DoWin1        START
              using GlobalData

              PushLong Win1Ptr
              _SelectWindow

              PushLong Win1Ptr
              _ShowWindow

              rts
              END

;*****
;
; DoWin2
;
; Selects and shows window 2 in response to menu selection.
;
;*****

DoWin2        START
              using GlobalData

              PushLong Win2Ptr
              _SelectWindow

              PushLong Win2Ptr
              _ShowWindow

              rts
              END

```

388 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

```
;*****
;
; DoWin3
;
; Selects and shows window 3 in response to menu selection.
;
;*****

DoWin3          START
                using GlobalData

                PushLong Win3Ptr
                _SelectWindow

                PushLong Win3Ptr
                _ShowWindow

                rts
                END

;*****
;
; DoClose
;
; Hides the active window when user presses button in its
; close box.
;
;*****

DoClose         START
                using GlobalData

                PushLong TaskData
                _HideWindow

                rts
                END

;*****
;
; Menu Data
;
;*****

MenuData        DATA

AppleMenu       dc c'>L@N1X',i1'13'
                dc c' LAbout this program...\N256',i1'13'
                dc c'.'

FileMenu        dc c'>L File \N2',i1'13'
                dc c' LQuit\N257*Qq',i1'13' ;Key command = Apple-Q
                dc c'.'

WindowsMenu     dc c'>L Windows \N3',i1'13'
                dc c' LWindow 1\N258',i1'13'
                dc c' LWindow 2\N259',i1'13'
                dc c' LWindow 3\N260',i1'13'
                dc c'.'

                END
```

```

;*****
;
; Window Data
;
;*****

WindowData      DATA

Win1ParamBlock anop
    dc    i'Win1End-Win1ParamBlock'
    dc    i2'%1101110110000000' Everything but info bar
    dc    i4'Win1Title'         Pointer to window's title
    dc    i4'0'                 Reserved (wRefCon)
    dc    i2'26,0,190,620'      Zoomed content region
    dc    i4'0'                 Default color table
    dc    i2'0'                 Vertical origin
    dc    i2'0'                 Horizontal origin
    dc    i2'200'               Data area height
    dc    i2'640'               Data area width
    dc    i2'200'               Max grow height
    dc    i2'640'               Max grow width
    dc    i2'4'                 Number of pixels to scroll vertically.
    dc    i2'16'               Number of pixels to scroll
horizontally.
    dc    i2'40'               Number of pixels to page vertically.
    dc    i2'160'              Number of pixels to page horizontally.
    dc    i4'0'                 Information bar text string.
    dc    i2'0'                 Info bar height
    dc    i4'0'                 Routine to draw shape (none, standard)
    dc    i4'0'                 Routine to draw info. bar.
    dc    i4'DrawWin1'          Routine to draw content.
    dc    i'40,20,100,360'      Size/pos of content
    dc    i4'-1'                Window's order (-1 means topmost)
    dc    i4'0'                Window Manager allocates wind. record
Win1End      anop

Win2ParamBlock anop
    dc    i'Win2End-Win2ParamBlock'
    dc    i2'%1101110110000000' Everything but info bar
    dc    i4'Win2Title'         Pointer to window's title
    dc    i4'0'                 Reserved (wRefCon)
    dc    i2'26,0,190,620'      Zoomed content region
    dc    i4'0'                 Default color table
    dc    i2'0'                 Vertical origin
    dc    i2'0'                 Horizontal origin
    dc    i2'200'               Data area height
    dc    i2'640'               Data area width
    dc    i2'200'               Max grow height
    dc    i2'640'               Max grow width
    dc    i2'4'                 Number of pixels to scroll vertically.
    dc    i2'16'               Number of pixels to scroll
horizontally.
    dc    i2'40'               Number of pixels to page vertically.
    dc    i2'160'              Number of pixels to page horizontally.
    dc    i4'0'                 Information bar text string.
    dc    i2'0'                 Info bar height
    dc    i4'0'                 Routine to draw shape (none, standard)
    dc    i4'0'                 Routine to draw info. bar.
    dc    i4'DrawWin2'          Routine to draw content.
    dc    i'50,30,110,380'      Size/pos of content
    dc    i4'-1'                Window's order (-1 means topmost)
    dc    i4'0'                Window Manager allocates wind. record
Win2End      anop

```

390 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

```

Win3ParamBlock anop
    dc    i'Win3End-Win3ParamBlock'
    dc    i2'*1101110110000000'    Everything but info bar
    dc    i4'Win3Title'              Pointer to window's title
    dc    i4'0'                      Reserved (wRefCon)
    dc    i2'26,0,190,620'           Zoomed content region
    dc    i4'0'                      Default color table
    dc    i2'0'                      Vertical origin
    dc    i2'0'                      Horizontal origin
    dc    i2'200'                    Data area height
    dc    i2'640'                    Data area width
    dc    i2'200'                    Max grow height
    dc    i2'640'                    Max grow width
    dc    i2'4'                      Number of pixels to scroll vertically.
    dc    i2'16'                    Number of pixels to scroll
horizontally.
    dc    i2'40'                    Number of pixels to page vertically.
    dc    i2'160'                   Number of pixels to page horizontally.
    dc    i4'0'                    Info bar text string.
    dc    i2'0'                    Info bar height
    dc    i4'0'                    Routine to draw shape (none, standard)
    dc    i4'0'                    Routine to draw info. bar.
    dc    i4'DrawWin3'              Routine to draw content.
    dc    i'60,40,120,400'          Size/pos of content
    dc    i4'-1'                   Window's order (-1 means topmost)
    dc    i4'0'                    Window Manager allocates wind. record
Win3End    anop

Win1Title   str 'Window 1'
Win2Title   str 'Window 2'
Win3Title   str 'Window 3'

PenPat1     dc h'11 11 11 11 11 11 11 11' ;Dithered blue pen pattern
            dc h'11 11 11 11 11 11 11 11'
            dc h'11 11 11 11 11 11 11 11'
            dc h'11 11 11 11 11 11 11 11'

Rect        dc i'20,20,40,40'        ;Rectangle painted in windows

; The rest of this data is for the dialog box.

MDialogPtr   ds 4

DRect        dc i'35,150,135,490'    ;BoundsRect of dialog window
OKBtnRect    dc i'85,280,0,0'        ;OK button display rect

TextRect1    dc i'15,10,35,500'
Text1        str 'As I said before:'

TextRect2    dc i'35,10,55,500'
Text2        str 'Copyright 1987 by Leo Scanlon'

ButtonText   str 'OK'

```

END

```

;*****
;
; Draw Windows
;
; Fills in the content region of each window, working rearward.
;
;*****

```



```
DrawWindows  START
              using GlobalData
```

```
    PushLong Win3Ptr          ;Window 3
    _SetPort
```

```
    jsl DrawWin3
```

```
    PushLong Win3Ptr
    _ShowWindow
```

```
    PushLong Win2Ptr          ;Window 2
    _SetPort
```

```
    jsl DrawWin2
```

```
    PushLong Win2Ptr
    _ShowWindow
```

```
    PushLong Win1Ptr          ;Window 1
    _SetPort
```

```
    jsl DrawWin1
```

```
    PushLong Win1Ptr
    _ShowWindow
```

```
    rts
    END
```

```
*****
;
; Draw Window 1
;
; Draws the contents of Window 1--a blue box on a white
; background. Called by DrawWindows initially and by the Window
; Manager when it updates the window.
;
*****
```

```
DrawWin1 START
              using WindowData
```

```
    PushWord #3                ;Background is white
    _SetSolidBackPat
```

```
    PushLong #PenPat1          ;Set pen pattern to blue
    _SetPenPat
```

```
    PushLong #Rect             ; and paint the rectangle
    _PaintRect
```

```
    rtl                        ;RTL required for update events
    END
```

392 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

```
;*****  
;  
; Draw Window 2  
;  
; Draws the contents of Window 2--a white box on a green  
; background. Called by DrawWindows initially and by the Window  
; Manager when it updates the window.  
;  
;*****
```

```
DrawWin2 START  
    using WindowData  
  
        PushWord #2                ;Background is green  
        _SetSolidBackPat  
  
        PushWord #3                ;Set pen pattern to white  
        _SetSolidPenPat  
  
        PushLong #Rect             ; and paint the rectangle  
        _PaintRect  
  
        rtl                        ;RTL required for update events  
    END
```

```
;*****  
;  
; Draw Window 3  
;  
; Draws the contents of Window 3--a green box on a red  
; background. Called by DrawWindows initially and by the Window  
; Manager when it updates the window.  
;  
;*****
```

```
DrawWin3 START  
    using WindowData  
  
        PushWord #1                _SetSolidBackPat  
  
        PushWord #2                ;Set pen pattern to green  
        _SetSolidPenPat  
  
        PushLong #Rect             ; and paint the rectangle  
        _PaintRect  
  
        rtl                        ;RTL required for update events  
    END
```

```
*****  
*  
* Prepare To Die  
*  
* Dies if carry is set.  
*  
* Error number to display is in X register.  
* Assumes that Miscellaneous Tools is active.  
*  
*****
```

```

PrepareToDie  START
               bcs RealDeath      ;Carry = 1?
               rts                 ; No. Return to caller

RealDeath     phx                 ; Yes. Goodbye, program.
               PushLong #DeathMsg
               _SysFailMgr

DeathMsg      str 'Could not handle error '

               END

;*****
;
; Mount Boot Disk
;
; Tells the user to insert the disk that contains the RAM-based
; tools.
;
;*****

MountBootDisk START

               _Set_Prefix SetPrefixParams
               _Get_Prefix GetPrefixParams

               PushWord #0          ;Space for result
               PushWord #195        ;Column position for dialog box
               PushWord #30         ;Row position for dialog box
               PushLong #PromptStr  ;Prompt at top of dialog box
               PushLong #VolStr     ;Volume name string
               PushLong #OKStr      ;String in Button 1
               PushLong #CancelStr  ;String in Button 2
               _TLMountVolume

               pla                  ;Obtain the button number
               rts                 ; and return to caller

PromptStr     str 'Please insert the disk.'
VolStr        ds 16
OKStr         str 'OK'
CancelStr     str 'Shutdown'

GetPrefixParams dc i'7'
               dc i4'VolStr'

SetPrefixParams dc i'7'
               dc i4'BootStr'

BootStr       str '*/'

               END

```

The box is quite simple; it contains only text and an OK button. The text can take either of two forms, however. The first time the user displays the box, it contains "Copyright 1987 by Leo Scanlon." After that, whenever

the user accesses the box, it shows not only the copyright notice, but also the clause “As I said before.”; this serves to illustrate how you can actually change dialog windows between appearances.

MODALD isn’t very different from MENUS, actually. Its GlobalData segment contains a NotFirstTime parameter that regulates what appears in the box. It’s set to 0 at the beginning of the program, then changed when the box first appears. GlobalData also contains the item types for the dialog (ButtonItem and StatText, in this case) and an ItemDisable that lets me disable the text.

The InitStuff segment contains start-up calls for the Line Editor and the Dialog Manager. Their shutdown calls are in the first segment, ModalD.) The item table in the DoMenu segment now includes the address of a DoAbout subroutine that brings up the dialog box and keeps it on the screen until the user presses OK.

DoAbout has what you might expect. It starts with a NewModalDialog call to create the box and NewDItem calls for the three items in it — a button and two lines of static text. (The data for these items is in the WindowData segment.) DoAbout ends with a ModalDialog call, which waits for the user to press OK, and a CloseDialog call to remove the box from the screen.

Tool Calls for Alert Boxes

Alert boxes are easiest of all to program, because they require only a single tool call. And that call does all the work of creating the box, interacting with the user, and closing the box. There are four calls that do these jobs: Alert, StopAlert, NoteAlert, and CautionAlert. As Table 12-4 shows, these calls differ only in what they display at the top left-hand corner. Alert displays nothing, while StopAlert, NoteAlert, and CautionAlert display an icon; a hand, a “talking” face, or an exclamation point, respectively (see Figure 12-6).



Stop



Note



Caution

Figure 12-6

Table 12-4

<u>Alert</u>		Draw general-purpose alert box
Call with:	PushWord #0 ;Space for result PushLong <i>AlertTemplate</i> ;Pointer to an alert template PushLong <i>filterProc</i> ;Pointer to filter used by ModalDialog <u>Alert</u>	
Result:	ID of item hit (word).	
Note:	See text for description of the alert template.	
<u>StopAlert</u>		Draw Stop alert box
Call with:	PushWord #0 ;Space for result PushLong <i>AlertTemplate</i> ;Pointer to an alert template PushLong <i>filterProc</i> ;Pointer to filter used by ModalDialog <u>StopAlert</u>	
Result:	ID of item hit (word).	
Note:	Same as Alert, but draws the Stop icon somewhere near the top left corner of the box.	
<u>NoteAlert</u>		Draw Note alert box
Call with:	PushWord #0 ;Space for result PushLong <i>AlertTemplate</i> ;Pointer to an alert template PushLong <i>filterProc</i> ;Pointer to filter used by ModalDialog <u>NoteAlert</u>	
Result:	ID of item hit (word).	
Note:	Same as Alert, but draws the Note icon somewhere near the top left corner of the box.	
<u>CautionAlert</u>		Draw Caution alert box
Call with:	PushWord #0 ;Space for result PushLong <i>AlertTemplate</i> ;Pointer to an alert template PushLong <i>filterProc</i> ;Pointer to filter used by ModalDialog <u>CautionAlert</u>	
Result:	ID of item hit (word).	
Note:	Same as Alert, but draws the Caution icon somewhere near the top left corner of the box.	

```

dc i1 '%10000010' ;Stage3
; (two beeps)
dc i1 '%10000011' ;Stage4
; (three beeps)
dc i4 'ButtonTemp' ; Pointer to
; button template
dc i4 'TextTemp' ;Pointer to text
; template
dc i4 '0' ;Terminator

```

Item Template

The item templates pointed to here are simply the inputs you use to set up dialog items with `NewDialogItem`, but in “template” form — that is, `itemID` (word), `itemRect` (rect), `itemType` (word), `itemDescr` (log), `itemValue` (word), `itemFlag` (word), and `itemColor` (long). For example, the `ButtonTemp` template mentioned earlier might look like this:

```
ButtonTemp dc i '1'           ; ID
           dc i '85,200,0,0'   ; Display rect
           dc i 'ButtonItem'    ; Simple button
           dc i 4 'ButtonText' ; Pointer to its text
           dc i '0'            ; Initial value
           dc i '0'            ; Default flag
           dc i 4 '0'          ; Default color table
```

Because everything is tucked neatly away in templates, the subroutine that displays the alert box can be quite short. If you have only a button enabled, for example (a typical situation with alert boxes), the subroutine might be:

```
PushWord #0           ; Space for result
PushLong #AlertTemp    ; Pointer to alert template
PushLong #0           ; No particular value
_StopAlert

pla                   ; It could only be one item.
rts                   ; Ignore it and leave.
```

Final Comments

Within these pages, I introduced you to the workings of the 65816 microprocessor and its role in the Apple IIGS. I also covered the Apple IIGS Programmer’s Workshop for assembly language and demonstrated how you can use tool sets within the IIGS Toolbox to do some common programming operations.

Where you go from here is, of course, up to you. It depends on what you want your programs to *do*. To create and play music, you need the services of the Sound Manager. To produce fancy, Macintosh-style lettering, you need the Font Manager. And so on. The point is that the Toolbox provides

tool sets and tools to do just about anything you want, and they're all well documented in the *Apple IIGS Toolbox Reference*. Moreover, if you can't find a tool set that meets some specific need, you can write one of your own! Details about doing that are also in the Toolbox Reference.

Well, all books must end somewhere, and this one ends here. I have genuinely enjoyed this journey through the resources of the Apple IIGS, and hope you have, too.

APPENDIX A

Hexadecimal/Decimal Conversion

HEXADECIMAL COLUMNS											
6		5		4		3		2		1	
HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC
0	0	0	0	0	0	0	0	0	0	0	0
1	1,048,576	1	65,536	1	4,096	1	256	1	16	1	1
2	2,097,152	2	131,072	2	8,192	2	512	2	32	2	2
3	3,145,728	3	196,608	3	12,288	3	768	3	48	3	3
4	4,194,304	4	262,144	4	16,384	4	1,024	4	64	4	4
5	5,242,880	5	327,680	5	20,480	5	1,280	5	80	5	5
6	6,291,456	6	393,216	6	24,576	6	1,536	6	96	6	6
7	7,340,032	7	458,752	7	28,672	7	1,792	7	112	7	7
8	8,388,608	8	524,288	8	32,768	8	2,048	8	128	8	8
9	9,437,184	9	589,824	9	36,864	9	2,304	9	144	9	9
A	10,485,760	A	655,360	A	40,960	A	2,560	A	160	A	10
B	11,534,336	B	720,896	B	45,056	B	2,816	B	176	B	11
C	12,582,912	C	786,432	C	49,152	C	3,072	C	192	C	12
D	13,631,488	D	851,968	D	53,248	D	3,328	D	208	D	13
E	14,680,064	E	917,504	E	57,344	E	3,584	E	224	E	14
F	15,728,640	F	983,040	F	61,440	F	3,840	F	240	F	15
7654		3210		7654		3210		7654		3210	
Byte				Byte				Byte			

POWERS OF 2

2^n	n
256	8
512	9
1,024	10
2,048	11
4,096	12
8,192	13
16,384	14
32,768	15
65,536	16
131,072	17
262,144	18
524,288	19
1,048,576	20
2,097,152	21
4,194,304	22
8,388,608	23
16,777,216	24

$2^0 = 16^0$
$2^4 = 16^1$
$2^8 = 16^2$
$2^{12} = 16^3$
$2^{16} = 16^4$
$2^{20} = 16^5$
$2^{24} = 16^6$
$2^{28} = 16^7$
$2^{32} = 16^8$
$2^{36} = 16^9$
$2^{40} = 16^{10}$
$2^{44} = 16^{11}$
$2^{48} = 16^{12}$
$2^{52} = 16^{13}$
$2^{56} = 16^{14}$
$2^{60} = 16^{15}$

POWERS OF 16

16^n	n
1	0
16	1
256	2
4,096	3
65,536	4
1,048,576	5
16,777,216	6
268,435,456	7
4,294,967,296	8
68,719,476,736	9
1,099,511,627,776	10
17,592,186,044,416	11
281,474,976,710,656	12
4,503,599,627,370,496	13
72,057,594,037,927,936	14
1,152,921,504,606,846,976	15

APPENDIX B

ASCII Table

MSD		0	1	2	3	4	5	6	7
LSD		000	001	010	011	100	101	110	111
0	0000	NUL	DLE	SP	0	@	P		p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	"	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	\$	4	D	T	d	t
5	0101	ENQ	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	'	7	G	W	g	w
8	1000	BS	CAN	(8	H	X	h	x
9	1001	HT	EM)	9	I	Y	i	y
A	1010	LF	SUB	*	:	J	Z	j	z
B	1011	VT	ESC	+	;	K	[k	{
C	1100	FF	FS	,	<	L	\	l	
D	1101	CR	GS	-	=	M]	m	}
E	1110	SO	RS	•	>	N	^	n	~
F	1111	SI	US	/	?	O	_	o	DEL

NUL — Null
 SOH — Start of Heading
 STX — Start of Text
 ETX — End of Text
 EOT — End of Transmission
 ENQ — Enquiry
 ACK — Acknowledge
 BEL — Bell
 BS — Backspace
 HT — Horizontal Tabulation
 LF — Line Feed
 VT — Vertical Tabulation
 FF — Form Feed
 CR — Carriage Return
 SO — Shift Out
 SI — Shift In

DLE — Data Link Escape
 DC — Device Control
 NAK — Negative Acknowledge
 SYN — Synchronous Idle
 ETB — End of Transmission Block
 CAN — Cancel
 EM — End of Medium
 SUB — Substitute
 ESC — Escape
 FS — File Separator
 GS — Group Separator
 RS — Record Separator
 US — Unit Separator
 SP — Space (Blank)
 DEL — Delete

65816 Instruction Set Summary

This appendix summarizes the 65816 instruction set in alphabetical order. It includes a short description of what each instruction does and which status flags it affects. Each entry also contains a table that lists the allowable addressing modes, the assembler format, the opcode, the number of bytes the instruction occupies in memory, and how long it takes to execute. The Assembler Format column includes the following abbreviations:

Num = 8- or 16-bit constant

Loc = 16-bit address

LongLoc = 24-bit address

DLoc = 16-bit address in the direct page

DLong = 24-bit address in the direct page

Dis8 = 8-bit signed relative displacement (distance forward or backward)

Dis16 = 16-bit signed relative displacement

Execution times are shown in clock cycles. To convert them to nanoseconds, multiply by 400 for a 2.5-MHz clock or by 1,000 for a 1-MHz clock. (Since 1,000 nanoseconds is 1 microsecond, you get microseconds if you use the cycle count directly.) The execution times are convenient for comparing the efficiency of one instruction or addressing mode with another. This helps you decide which approach is best, in case you have doubts.

ADC*Add to Accumulator with Carry*

Operation: Add the operand and the carry flag to the accumulator.

N	V	M	X	D	I	Z	C
*	*	*	*

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Immediate	ADC #Num	69	3 ²	3 ³
Absolute	ADC Loc	6D	3	5 ³
Absolute long	ADC LongLoc	6F	4	6 ³
Absolute indexed with X	ADC Loc,X	7D	3	5 ^{1,3}
Absolute long indexed with X	ADC LongLoc,X	7F	4	6 ³
Absolute indexed with Y	ADC Loc,Y	79	3	5 ^{1,3}
Direct	ADC DLoc	65	2	4 ^{3,4}
Direct indexed with X	ADC DLoc,X	75	2	5 ^{3,4}
Direct indirect	ADC (DLoc)	72	2	6 ^{3,4}
Direct indirect long	ADC [DLong]	67	2	7 ^{3,4}
Direct indirect indexed	ADC (DLoc),Y	71	2	6 ^{1,3,4}
Direct indirect indexed long	ADC [DLong],Y	77	2	7 ^{3,4}
Direct indexed indirect	ADC (DLoc,X)	61	2	7 ^{3,4}
Stack relative	ADC Dis8,S	63	2	5 ³
Stack relative indirect indexed	ADC (Dis8,S),Y	73	2	8 ³

¹Add 1 cycle if adding index crosses a page boundary.²Subtract 1 byte if M = 1 (8-bit memory/accumulator).³Subtract 1 cycle if M = 1 (8-bit memory/accumulator).⁴Add 1 cycle if direct register low (DL) is not equal to 0.

AND*AND Memory with Accumulator*

Operation: ANDs the operand with the accumulator.

N	V	M	X	D	I	Z	C
*	*	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Immediate	AND #Num	29	3 ²	3 ³
Absolute	AND Loc	2D	3	5 ³
Absolute long	AND LongLoc	2F	4	6 ³
Absolute indexed with X	AND Loc,X	3D	3	5 ^{1,3}
Absolute long indexed with X	AND LongLoc,X	3F	4	6 ³
Absolute indexed with Y	AND Loc,Y	39	3	5 ^{1,3}
Direct	AND DLoc	25	2	4 ^{3,4}
Direct indexed with X	AND DLoc,X	35	2	5 ^{3,4}
Direct indirect	AND (DLoc)	32	2	6 ^{3,4}
Direct indirect long	AND [DLong]	27	2	7 ^{3,4}
Direct indirect indexed	AND (DLoc),Y	31	2	6 ^{1,3,4}
Direct indirect indexed long	AND [DLong],Y	37	2	7 ^{3,4}
Direct indexed indirect	AND (DLoc,X)	21	2	7 ^{3,4}
Stack relative	AND Dis8,S	23	2	5 ³
Stack relative indirect indexed	AND (Dis8,S),Y	33	2	8 ³

¹Add 1 cycle if adding index crosses a page boundary.

²Subtract 1 byte if M = 1 (8-bit memory/accumulator).

³Subtract 1 cycle if M = 1 (8-bit memory/accumulator).

⁴Add 1 cycle if direct register low (DL) is not equal to 0.

ASL*Arithmetic Shift Left*

Operation: Shifts the operand left one bit position. It puts the displaced bit in the carry flag (C) and puts a 0 in the vacated bit position. See also LSR, which shifts operands right.

N	V	M	X	D	I	Z	C
*	*	*

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Accumulator	ASL A	0A	1	2
Absolute	ASL Loc	0E	3	8 ²
Absolute indexed with X	ASL Loc,X	1E	3	9 ^{1,2}
Direct	ASL DLoc	06	2	7 ^{2,3}
Direct indexed with X	ASL DLoc,X	16	2	8 ^{2,3}

¹Add 1 cycle if adding index crosses a page boundary.

²Subtract 2 cycles if M = 1 (8-bit memory/accumulator).

³Add 1 cycle if direct register low (DL) is not equal to 0.

BCC
BLT*Branch on Carry Clear*
Branch if Less Than

Operation: Branches on C = 0.

N	V	M	X	D	I	Z	C
.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Program counter relative	BCC Dis8	90	2	2 ¹

¹Add 1 cycle if branch is taken. In emulation mode only, add 1 additional cycle if the branch taken is to a different page.

BCS
BGE*Branch on Carry Set*
Branch if Greater Than or Equal

Operation: Branches on C = 1.

N	V	M	X	D	I	Z	C
.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Program counter relative	BCS Dis8	B0	2	2 ¹

¹Add 1 cycle if branch is taken. In emulation mode only, add 1 additional cycle if the branch taken is a different page.

BEQ*Branch if Equal*

Operation: Branches on Z = 1.

N	V	M	X	D	I	Z	C
.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Program counter relative	BEQ Dis8	F0	2	2 ¹

¹Add 1 cycle if branch is taken. In emulation mode only, add 1 additional cycle if the branch taken is to a different page.

BGE — See **BCS**

BIT*Bit Test*

Operation: ANDs the operand with the accumulator, but reports the result only in the flags.
The operands are unaffected.

	N	V	M	X	D	I	Z	C
Immediate mode	*	.
Other modes, 8-bit	M ⁷	M ⁶	*	.
Other modes, 16-bit	M ¹⁵	M ¹⁴	*	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Immediate	BIT #Num	89	3 ²	3 ³
Absolute	BIT Loc	2C	3	5 ³
Absolute indexed with X	BIT Loc,X	3C	3	5 ^{1,3}
Direct	BIT DLoc	24	2	4 ^{3,4}
Direct indexed with X	BIT DLoc,X	34	2	5 ^{3,4}

¹Add 1 cycle if adding index crosses a page boundary.

²Subtract 1 byte if M = 1 (8-bit memory/accumulator).

³Subtract 1 cycle if M = 1 (8-bit memory/accumulator).

⁴Add 1 cycle if direct register low (DL) is not equal to 0.

BLT — See **BCC**

BMI*Branch if Minus*

Operation: Branches on N = 1.

N	V	M	X	D	I	Z	C
.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Program counter relative	BMI Dis8	30	2	2 ¹

¹Add 1 cycle if branch is taken. In emulation mode only, add 1 additional cycle if the branch taken is to a different page.

BNE*Branch if Not Equal*

Operation: Branches on Z = 0.

N	V	M	X	D	I	Z	C
.

Addressing Mode	Assembler Format		Opcode	Bytes	Cycles
Program counter relative	BNE	Dis8	D0	2	2 ¹

¹Add 1 cycle if branch is taken. In emulation mode only, add 1 additional cycle if the branch taken is to a different page.

BPL*Branch if Plus*

Operation: Branches on N = 0.

N	V	M	X	D	I	Z	C
.

Addressing Mode	Assembler Format		Opcode	Bytes	Cycles
Program counter relative	BPL	Dis8	10	2	2 ¹

¹Add 1 cycle if branch is taken. In emulation mode only, add 1 additional cycle if the branch taken is a different page.

BRA*Branch Always*Operation: Branches unconditionally. See also **BRL**.

N	V	M	X	D	I	Z	C
.

Addressing Mode	Assembler Format		Opcode	Bytes	Cycles
Program counter relative	BRA	Dis8	80	2	2 ¹

¹Add 1 cycle if branch is taken. In emulation mode only, add 1 additional cycle if the branch taken is to a different page.

BRK*Force Break*

Operation: Forces a software interrupt.

In emulation mode, BRK adds 2 to the program counter and pushes it onto the stack; sets the break (B) flag in the status register and pushes it; and loads the program counter with the address in locations \$00FFFE and \$00FFFF.

In native mode, BRK pushes the program bank register onto the stack; adds 2 to the program counter and pushes it onto the stack; pushes the status register; clears the program bank register; and loads the program counter with the address in locations \$00FFE6 and \$00FFE7.

<i>Emulation mode</i>								<i>Native mode</i>							
N	V	I	B	D	I	Z	C	N	V	M	X	D	I	Z	C
.	.	.	1	0	1	0	1	.	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Stack	BRK or BRK nn	00	2 ¹	8 ²

¹BRK is a 1-byte instruction, but the program counter is incremented by 2, thereby allowing for a 1-byte identifier.

²Subtract 1 cycle in emulation mode (E = 1).

BRL*Branch Always Long*Operation: Branches unconditionally to any location in a 64K memory bank. See also **BRA**.

N	V	M	X	D	I	Z	C
.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Program counter relative long	BRL Dis16	82	3	3

BVC*Branch on Overflow Clear*

Operation: Branches on V = 0.

N	V	M	X	D	I	Z	C
.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Program counter relative	BVC Dis8	50	2	2 ¹

¹Add 1 cycle if branch is taken. In emulation mode only, add 1 additional cycle if the branch taken is to a different page.

BVS*Branch on Overflow Set*

Operation: Branches on V = 1.

N	V	M	X	D	I	Z	C
.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Program counter relative	BVS Dis8	70	2	2 ¹

¹Add 1 cycle if branch is taken. In emulation mode only, add 1 additional cycle if the branch taken is to a different page.

CLC*Clear Carry Flag*

Operation: C = 0.

N	V	M	X	D	I	Z	C
.	0

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Implied	CLC	18	1	2

CLD*Clear Decimal Mode*

Operation: Sets D = 0, to select the binary mode.

N	V	M	X	D	I	Z	C
.	.	.	.	0	.	.	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Implied	CLD	D8	1	2

¹Add 1 cycle if branch is taken. In emulation mode only, add 1 additional cycle if the branch taken is to a different page.

CLI*Clear Interrupt Disable Bit*

Operation: Sets I = 0, to enable interrupts.

N	V	M	X	D	I	Z	C
.	0	.	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Implied	CLI	58	1	2

CLV*Clear Overflow Flag*Operation: $V = 0$.

N	V	M	X	D	I	Z	C
.	0

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Implied	CLV	B8	1	2

CMA — See CMP**CMP (or CMA)***Compare Memory and Accumulator*

Operation: Compares by subtracting the operand from the accumulator, but reports the result only in the flags. The accumulator is unaffected.

N	V	M	X	D	I	Z	C
*	*	*

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Immediate	CMP #Num	C9	3 ²	3 ³
Absolute	CMP Loc	CD	3	5 ³
Absolute long	CMP LongLoc	CF	4	6 ³
Absolute indexed with X	CMP Loc,X	DD	3	5 ^{1,3}
Absolute long indexed with X	CMP LongLoc,X	DF	4	6 ³
Absolute indexed with Y	CMP Loc,Y	D9	3	5 ^{1,3}
Direct	CMP DLoc	C5	2	4 ^{3,4}
Direct indexed with X	CMP DLoc,X	D5	2	5 ^{3,4}
Direct indirect	CMP (DLoc)	D2	2	6 ^{3,4}
Direct indirect long	CMP [DLong]	C7	2	7 ^{3,4}
Direct indirect indexed	CMP (DLoc),Y	D1	2	6 ^{1,3,4}
Direct indirect indexed long	CMP [DLong],Y	D7	2	7 ^{3,4}
Direct indexed indirect	CMP (DLoc,X)	C1	2	7 ^{3,4}
Stack relative	CMP Dis8,S	C3	2	5 ³
Stack relative indirect indexed	CMP (Dis8,S),Y	D3	2	8 ³

¹Add 1 cycle if adding index crosses a page boundary.²Subtract 1 byte if M = 1 (8-bit memory/accumulator).³Subtract 1 cycle if M = 1 (8-bit memory/accumulator).⁴Add 1 cycle if direct register low (DL) is not equal to 0.

COP*Coprocessor*

Operation: Forces an interrupt to the vector at locations 00FFE4 and 00FFE5 (native mode) or 00FFF4 and 00FFF5 (emulation mode).

N	V	M	X	D	I	Z	C
.	.	.	.	0	1	.	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Stack	COP nn	02	2	8 ¹

¹Subtract 1 cycle in emulation mode.

CPX*Compare Memory and X Register*

Operation: Compares by subtracting the operand from the X register, but reports the result only in the flags. X is unaffected.

N	V	M	X	D	I	Z	C
*	*	*

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Immediate	CPX #Num	E0	3 ¹	3 ²
Absolute	CPX Loc	ECX	3	5 ²
Direct	CPX DLoc	E4	2	4 ^{2,3}

¹Subtract 1 byte if X = 1 (8-bit index registers).

²Subtract 1 cycle if X = 1 (8-bit index registers).

³Add 1 cycle if direct register low (DL) is not equal to 0.

CPY*Compare Memory and Y Register*

Operation: Compares by subtracting the operand from the Y register, but reports the result only in the flags. Y is unaffected.

N	V	M	X	D	I	Z	C
*	*	*

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Immediate	CPY #Num	C0	3 ¹	3 ²
Absolute	CPY Loc	CC	3	5 ²
Direct	CPY DLoc	C4	2	4 ^{2,3}

¹Subtract 1 byte if X = 1 (8-bit index registers).

²Subtract 1 cycle if X = 1 (8-bit index registers).

³Add 1 cycle if direct register low (DL) is not equal to 0.

DEC*Decrement Memory or Accumulator*

Operation: Subtracts 1 from the operand.

N	V	M	X	D	I	Z	C
*	*	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Accumulator	DEC A or DEA	3A	1	2
Absolute	DEC Loc	CE	3	8 ²
Absolute indexed with X	DEC Loc,X	DE	3	9 ^{1,2}
Direct	DEC DLoc	C6	2	7 ^{2,3}
Direct indexed with X	DEC DLoc,X	D6	2	8 ^{2,3}

¹Add 1 cycle if adding index crosses a page boundary.²Subtract 2 cycles if M = 1 (8-bit memory/accumulator).³Add 1 cycle if direct register low (DL) is not equal to 0.**DEX***Decrement X Register*

Operation: Subtracts 1 from the X register.

N	V	M	X	D	I	Z	C
*	*	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Implied	DEX	CA	1	2

DEY*Decrement Y Register*

Operation: Subtracts 1 from the Y register.

N	V	M	X	D	I	Z	C
*	*	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Implied	DEY	88	1	2

EOR*Exclusive-OR Memory with Accumulator*

Operation: Exclusive-ORs the operand with the accumulator.

N	V	M	X	D	I	Z	C
*	*	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Immediate	EOR #Num	49	3 ²	3 ³
Absolute	EOR Loc	4D	3	5 ³
Absolute long	EOR LongLoc	4F	4	6 ³
Absolute indexed with X	EOR Loc,X	5D	3	5 ^{1,3}
Absolute long indexed with X	EOR LongLoc,X	5F	4	6 ³
Absolute indexed with Y	EOR Loc,Y	59	3	5 ^{1,3}
Direct	EOR DLoc	45	2	4 ^{3,4}
Direct indexed with X	EOR DLoc,X	55	2	5 ^{3,4}
Direct indirect	EOR (DLoc)	52	2	6 ^{3,4}
Direct indirect long	EOR [DLong]	47	2	7 ^{3,4}
Direct indirect indexed	EOR (DLoc),Y	51	2	6 ^{1,3,4}
Direct indirect indexed long	EOR [DLong],Y	57	2	7 ^{3,4}
Direct indexed indirect	EOR (DLoc,X)	41	2	7 ^{3,4}
Stack relative	EOR Dis8,S	43	2	5 ³
Stack relative indirect indexed	EOR (Dis8,S),Y	53	2	8 ³

¹Add 1 cycle if adding index crosses a page boundary.²Subtract 1 byte if M = 1 (8-bit memory/accumulator).³Subtract 1 cycle if M = 1 (8-bit memory/accumulator).⁴Add 1 cycle if direct register low (DL) is not equal to 0.**INC***Increment Memory or Accumulator*

Operation: Adds 1 to the operand.

N	V	M	X	D	I	Z	C
*	*	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Accumulator	INC A or INA	1A	1	2
Absolute	INC Loc	EE	3	8 ²
Absolute indexed with X	INC Loc,X	FE	3	9 ^{1,2}
Direct	INC DLoc	E6	2	7 ^{2,3}
Direct indexed with X	INC DLoc,X	F6	2	8 ^{2,3}

¹Add 1 cycle if adding index crosses a page boundary.²Subtract 2 cycles if M = 1 (8-bit memory/accumulator).³Add 1 cycle if direct register low (DL) is not equal to 0.

INX*Increment X Register*

Operation: Adds 1 to the X register.

N	V	M	X	D	I	Z	C
*	*	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Implied	INX	E8	1	2

INY*Increment Y Register*

Operation: Adds 1 to the Y register.

N	V	M	X	D	I	Z	C
*	*	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Implied	INY	C8	1	2

JML*Jump Long*Operation: Transfers to a new location indirectly, using an address it obtains from a 3-byte pointer in memory. See also **JMP**.

N	V	M	X	D	I	Z	C
.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Absolute indirect	JML (Loc)	DC	3	6

JMP*Jump*

Operation: Transfers to the specified target location. Note that JMP's absolute indirect mode obtains a 2-byte address (PC contents) from the indirect location. Compare this with JML, which obtains a 3-byte address (PC plus PBR) from the location.

N V M X D I Z C

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Absolute	JMP Loc	4C	3	3
Absolute long	JMP LongLoc	5C	4	4
Absolute indirect	JMP (Loc)	6C	3	5
Absolute indexed indirect	JMP (Loc,X)	7C	3	6

JSL*Jump to Subroutine Long*

Operation: Saves the PC and program bank register (PBR) on the stack, then transfers to the specified target location. The target can be anywhere in memory. Compare JSL with JSR, which can only transfer within the current bank. An RTL instruction is used to return from the subroutine.

N V M X D I Z C

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Absolute long	JSL LongLoc	22	4	8

JSR*Jump to Subroutine*

Operation: Saves the PC on the stack, then transfers to the specified target location. The target must be within the current program bank. Compare JSR with JSL, which can transfer anywhere in memory. An RTS instruction is used to return from the subroutine.

N V M X D I Z C

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Absolute	JSR Loc	20	3	6
Absolute indexed indirect	JSR (DLoc,X)	FC	3	6

LDA*Load Accumulator*

Operation: Loads the accumulator with the contents of the operand.

N	V	M	X	D	I	Z	C
*	*	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Immediate	LDA #Num	A9	3 ²	3 ³
Absolute	LDA Loc	AD	3	5 ³
Absolute long	LDA LongLoc	AF	4	6 ³
Absolute indexed with X	LDA Loc,X	BD	3	5 ^{1,3}
Absolute long indexed with X	LDA LongLoc,X	BF	4	6 ³
Absolute indexed with Y	LDA Loc,Y	B9	3	5 ^{1,3}
Direct	LDA DLoc	A5	2	4 ^{3,4}
Direct indexed with X	LDA DLoc,X	B5	2	5 ^{3,4}
Direct indirect	LDA (DLoc)	B2	2	6 ^{3,4}
Direct indirect indexed	LDA (DLoc),Y	B1	2	6 ^{1,3,4}
Direct indirect indexed long	LDA [DLong],Y	B7	2	7 ^{3,4}
Direct indexed indirect	LDA (DLoc,X)	A1	2	7 ^{3,4}
Stack relative	LDA Dis8,S	A3	2	5 ³
Stack relative indirect indexed	LDA (Dis8,S),Y	B3	2	8 ³

¹Add 1 cycle if adding index crosses a page boundary.²Subtract 1 byte if M = 1 (8-bit memory/accumulator).³Subtract 1 cycle if M = 1 (8-bit memory/accumulator).⁴Add 1 cycle if direct register low (DL) is not equal to 0.**LDX***Load X register*

Operation: Loads the X register with the contents of the operand.

N	V	M	X	D	I	Z	C
*	*	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Immediate	LDX #Num	A2	2 ²	3 ³
Absolute	LDX Loc	AE	3	5 ³
Absolute indexed with Y	LDX Loc,Y	BE	3	5 ^{1,3}
Direct	LDX DLoc	A6	2	4 ^{3,4}
Direct indexed with Y	LDX DLoc,Y	B6	2	5 ^{3,4}

¹Add 1 cycle if adding index crosses a page boundary.²Subtract 1 byte if X = 1 (8-bit index registers).³Subtract 1 cycle if X = 1 (8-bit index registers).⁴Add 1 cycle if direct register low (DL) is not equal to 0.

LDY*Load Y register*

Operation: Loads the Y register with the contents of the operand.

N	V	M	X	D	I	Z	C
*	*	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Immediate	LDY #Num	A0	3 ²	3 ³
Absolute	LDY Loc	AC	3	5 ³
Absolute indexed with X	LDY Loc,X	BC	3	5 ^{1,3}
Direct	LDY DLoc	A4	2	4 ^{3,4}
Direct indexed with X	LDY DLoc,X	B4	2	5 ^{3,4}

¹Add 1 cycle if adding index crosses a page boundary.²Subtract 1 byte if X = 1 (8-bit index registers).³Subtract 1 cycle if X = 1 (8-bit index registers).⁴Add 1 cycle if direct register low (DL) is not equal to 0.**LSR***Logical Shift Right*

Operation: Shifts the operand right one bit position. It puts the displaced bit in the carry flag (C) and puts a 0 in the vacated bit position. See also ASL, which shifts operands left.

N	V	M	X	D	I	Z	C
0	*	*

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Accumulator	LSR A	4A	1	2
Absolute	LSR Loc	4E	3	8 ²
Absolute indexed with X	LSR Loc,X	5E	3	9 ^{1,2}
Direct	LSR DLoc	46	2	7 ^{2,3}
Direct indexed with X	LSR DLoc,X	56	2	8 ^{2,3}

¹Add 1 cycle if adding index crosses a page boundary.²Subtract 2 cycles if M = 1 (8-bit memory/accumulator).³Add 1 cycle if direct register low (DL) is not equal to 0.

MVN*Move Block Negative*

Operation: Moves a block of bytes to a lower-numbered address. Before executing MVN, set up the following registers:

- X = Address of beginning of source block
- Y = Address of beginning of destination block
- A = Number of bytes to be copied, less 1

N V M X D I Z C

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Block move	MVN ss,DLoc	54	3	7*n

Here, *ss* and *DLoc* are the hexadecimal numbers of the source and destination banks, while *n* is the number of bytes moved.

MVP*Move Block Positive*

Operation: Moves a block of bytes to a higher-numbered address. Before executing MVP, set up the following registers:

- X = Address of end of source block
- Y = Address of end of destination block
- A = Number of bytes to be copied, less 1

N V M X D I Z C

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Block move	MVP ss,DLoc	44	3	7*n

Here, *ss* and *DLoc* are the hexadecimal numbers of the source and destination banks, while *n* is the number of bytes moved.

NOP*No Operation*

Operation: Does nothing except advance the program counter.

N V M X D I Z C

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Implied	NOP	EA	1	2

ORA*OR Memory with Accumulator*

Operation: ORs the operand with the accumulator.

N	V	M	X	D	I	Z	C
*	*	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Immediate	ORA #Num	09	3 ²	3 ³
Absolute	ORA Loc	0D	3	5 ³
Absolute long	ORA LongLoc	0F	4	6 ³
Absolute indexed with X	ORA Loc,X	1D	3	5 ^{1,3}
Absolute long indexed with X	ORA LongLoc,X	1F	4	6 ³
Absolute indexed with Y	ORA Loc,Y	19	3	5 ^{1,3}
Direct	ORA DLoc	05	2	4 ^{3,4}
Direct indexed with X	ORA DLoc,X	15	2	5 ^{3,4}
Direct indirect	ORA (DLoc)	12	2	6 ^{3,4}
Direct indirect long	ORA [DLoc]	07	2	7 ³
Direct indirect indexed	ORA (DLoc),Y	11	2	6 ^{1,3,4}
Direct indirect indexed long	ORA [DLoc),Y	17	2	7 ^{3,4}
Direct indexed indirect	ORA (DLoc,X)	01	2	7 ^{3,4}
Stack relative	ORA Dis8,S	03	2	5 ³
Stack relative indirect indexed	ORA (Dis8,S),Y	13	2	8 ³

¹Add 1 cycle if adding index crosses a page boundary.²Subtract 1 byte if M = 1 (8-bit memory/accumulator).³Subtract 1 cycle if M = 1 (8-bit memory/accumulator).⁴Add 1 cycle if direct register low (DL) is not equal to 0.**PEA***Push Effective Absolute Address onto Stack
or Push Immediate Word onto Stack*

Operation: Pushes a 16-bit operand onto the stack.

N	V	M	X	D	I	Z	C
.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Stack	PEA Loc	F4	3	5

PEI

*Push Effective Indirect Address onto Stack
or Push Direct Page Word onto Stack*

Operation: Adds the offset to the direct page register, then pushes the 2 bytes at that address onto the stack.

N V M X D I Z C
· · · · · · · ·

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Stack	PEI (DLoc)	D4	2	6 ¹

¹Add 1 cycle if data register low (DL) is not equal to 0.

PER

Push Effective Program Counter Relative Address onto Stack

Operation: Adds a 16-bit constant to the program counter, then pushes the resulting value onto the stack.

N V M X D I Z C
· · · · · · · ·

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Stack	PER #Num	62	3	6

PHA

Push Accumulator onto Stack

Operation: Pushes A onto the stack.

N V M X D I Z C
· · · · · · · ·

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Stack	PHA	48	1	4 ¹

PHB

Push Data Bank Register onto Stack

Operation: Pushes the DBR onto the stack.

N V M X D I Z C
· · · · · · · ·

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Stack	PHB	8B	1	3

420 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

PHD

Push Direct Register onto Stack

Operation: Pushes the D register onto the stack.

N V M X D I Z C
· · · · · · ·

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Stack	PHD	0B	1	3

PHK

Push Program Bank Register onto Stack

Operation: Pushes the PBR onto the stack.

N V M X D I Z C
· · · · · · ·

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Stack	PHK	4B	1	3

PHP

Push Processor Status Register onto Stack

Operation: Pushes the P register onto the stack.

N V M X D I Z C
· · · · · · ·

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Stack	PHP	08	1	3

PHX

Push X Register onto Stack

Operation: Pushes X onto the stack.

N V M X D I Z C
· · · · · · ·

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Stack	PHX	DA	1	4 ¹

¹Subtract 1 cycle if X = 1 (8-bit index registers).

PHY*Push Y Register onto Stack*

Operation: Pushes Y onto the stack.

N	V	M	X	D	I	Z	C
.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Stack	PHY	5A	1	4 ¹

¹Subtract 1 cycle if X = 1 (8-bit index registers).**PLA***Pull Accumulator from Stack*

Operation: Retrieves A from the stack.

N	V	M	X	D	I	Z	C
*	*	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Stack	PLA	68	1	5 ¹

PLB*Pull Data Bank Register from Stack*

Operation: Retrieves the DBR from the stack.

N	V	M	X	D	I	Z	C
*	*	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Stack	PHB	AB	1	4

PLD*Pull Direct Register from Stack*

Operation: Retrieves D from the stack.

N	V	M	X	D	I	Z	C
*	*	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Stack	PLD	2B	1	4

PLP*Pull Processor Status Register from Stack*

Operation: Retrieves P from the stack.

N	V	M	X	D	I	Z	C
*	*	*	*	*	*	*	*

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Stack	PLP	28	1	4

PLX*Pull X Register from Stack*

Operation: Retrieves X from the stack.

N	V	M	X	D	I	Z	C
*	*	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Stack	PLX	FA	1	5 ¹

PLY*Pull Y Register from Stack*

Operation: Retrieves Y from the stack.

N	V	M	X	D	I	Z	C
*	*	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Stack	PLY	7A	1	5 ¹

REP*Reset Processor Status Bits*

Operation: ANDs the processor status register with the complement of an immediate byte.
Thus, each 1 bit in the operand clears the corresponding bit in P.

	N	V	M	X	D	I	Z	C
(Native mode)	*	*	*	*	*	*	*	*
(Emulation mode)	*	*	.	.	*	*	*	*

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Immediate	REP #Num	C2	2	3

ROL*Rotate Left*

Operation: Rotates the operand left one bit position. It puts the contents of the carry flag (C) in the vacated bit position, then puts the displaced bit in C. See also **ROR**, which rotates operands right.

N	V	M	X	D	I	Z	C
*	*	*

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Accumulator	ROL A	2A	1	2
Absolute	ROL Loc	2E	3	8 ²
Absolute indexed with X	ROL Loc,X	3E	3	9 ^{1,2}
Direct	ROL DLoc	26	2	7 ^{2,3}
Direct indexed with X	ROL DLoc,X	36	2	8 ^{2,3}

¹Add 1 cycle if adding index crosses a page boundary.

²Subtract 2 cycles if M = 1 (8-bit memory/accumulator).

³Add 1 cycle if direct register low (DL) is not equal to 0.

ROR*Rotate Right*

Operation: Rotates the operand right one bit position. It puts the contents of the carry flag (C) in the vacated bit position, then puts the displaced bit in C. See also **ROL**, which rotates operands left.

N	V	M	X	D	I	Z	C
*	*	*

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Accumulator	ROR A	6A	1	2
Absolute	ROR Loc	6E	3	8 ²
Absolute indexed with X	ROR Loc,X	7E	3	9 ^{1,2}
Direct	ROR DLoc	66	2	7 ²
Direct indexed with X	ROR DLoc,X	76	2	8 ^{2,3}

¹Add 1 cycle if adding index crosses a page boundary.

²Subtract 2 cycles if M = 1 (8-bit memory/accumulator).

³Add 1 cycle if direct register low (DL) is not equal to 0.

RTI*Return from Interrupt*

Operation: Retrieves the contents of the processor status register (P), program counter (PC), and program bank register (PBR) from the stack.

N V M X D I Z C
* * * * *

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Stack	RTI	40	1	7

RTL*Return from Subroutine Long*

Operation: Retrieves the contents of the program counter (PC) and program bank register (PBR) from the stack. That is, it undoes the work of a JSL instruction.

N V M X D I Z C
.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Stack	RTL	6B	1	6

RTS*Return from Subroutine*

Operation: Retrieves the contents of the program counter (PC) from the stack. That is, it undoes the work of a JSR instruction.

N V M X D I Z C
.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Stack	RTS	60	1	6

SBC*Subtract from Accumulator with Borrow*

Operation: Subtract the operand and the complement of the carry flag from the accumulator.

In effect, the operation is $A = A - M + C$.

N	V	M	X	D	I	Z	C
*	*	*	*

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Immediate	SBC #Num	E9	3 ²	3 ³
Absolute	SBC Loc	ED	3	5 ³
Absolute long	SBC LongLoc	EF	4	6 ³
Absolute indexed with X	SBC Loc,X	FD	3	5 ^{1,3}
Absolute long indexed with X	SBC LongLoc,X	FF	4	6 ³
Absolute indexed with Y	SBC Loc,Y	F9	3	5 ^{1,3}
Direct	SBC DLoc	E5	2	4 ^{3,4}
Direct indexed with X	SBC DLoc,X	F5	2	5 ^{3,4}
Direct indirect	SBC (DLoc)	F2	2	6 ^{3,4}
Direct indirect long	SBC [DLong]	E7	2	7 ^{3,4}
Direct indirect indexed	SBC (DLoc),Y	F1	2	6 ^{1,3,4}
Direct indirect indexed long	SBC [DLong],Y	F7	2	7 ^{3,4}
Direct indexed indirect	SBC (DLoc,X)	E1	2	7 ^{3,4}
Stack relative	SBC Dis8,S	E3	2	5 ³
Stack relative indirect indexed	SBC (Dis8,S),Y	F3	2	8 ³

¹Add 1 cycle if adding index crosses a page boundary.²Subtract 1 byte if M = 1 (8-bit memory/accumulator).³Subtract 1 cycle if M = 1 (8-bit memory/accumulator).⁴Add 1 cycle if direct register low (DL) is not equal to 0.**SEC***Set Carry Flag*

Operation: C = 1.

N	V	M	X	D	I	Z	C
.	1

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Implied	SEC	38	1	2

SED*Set Decimal Mode*

Operation: Sets D = 1, to select the decimal mode.

N	V	M	X	D	I	Z	C
.	.	.	.	1	.	.	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Implied	SED	F8	1	2

SEI*Set Interrupt Disable Bit*

Operation: Sets I = 1, to disable or lock out interrupts.

N	V	M	X	D	I	Z	C
.	1	.	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Implied	SEI	78	1	2

SEP*Set Processor Status Bits*

Operation: ORs the processor status register with an immediate byte. Thus, each 1 bit in the operand sets the corresponding bit in P.

	N	V	M	X	D	I	Z	C
(Native mode)	*	*	*	*	*	*	*	*
(Emulation mode)	*	*	.	.	*	*	*	*

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Immediate	SEP #nn	E2	2	3

STA*Store Accumulator*

Operation: Stores the contents of the accumulator at the specified memory location.

N	V	M	X	D	I	Z	C
.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Absolute	STA Loc	8D	3	5 ²
Absolute long	STA LongLoc	8F	4	6 ²
Absolute indexed with X	STA Loc,X	9D	3	6 ^{1,2}
Absolute long indexed with X	STA LongLoc,X	9F	4	6 ²
Absolute indexed with Y	STA Loc,Y	99	3	6 ^{1,2}
Direct	STA DLoc	85	2	4 ^{2,3}
Direct indexed with X	STA DLoc,X	85	2	5 ^{2,3}
Direct indirect	STA (DLoc)	92	2	6 ^{2,3}
Direct indirect long	STA [DLong]	87	2	7 ^{2,3}
Direct indirect indexed	STA (DLoc),Y	91	2	7 ^{1,2,3}
Direct indirect indexed long	STA [DLong],Y	97	2	7 ^{2,3}
Direct indexed indirect	STA (DLoc,X)	81	2	7 ^{2,3}
Stack relative	STA Dis8,S	83	2	5 ²
Stack relative indirect indexed	STA (Dis8,S),Y	93	2	8 ²

¹Add 1 cycle if adding index crosses a page boundary.²Subtract 1 cycle if M = 1 (8-bit memory/accumulator).³Add 1 cycle if direct register low (DL) is not equal to 0.**STP***Stop the Clock*

Operation: Stops the processor until it receives a hardware reset.

N	V	M	X	D	I	Z	C
.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Implied	STP	DB	1	3

STX*Store X Register*

Operation: Stores the contents of the X register at the specified memory location.

N	V	M	X	D	I	Z	C
.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Absolute	STX Loc	8E	3	5 ¹
Direct	STX DLoc	86	2	4 ^{1,2}
Direct indexed with Y	STX DLoc,Y	96	2	5 ¹

¹Subtract 1 cycle if X = 1 (8-bit index registers).²Add 1 cycle if direct register low (DL) is not equal to 0.**STY***Store Y Register*

Operation: Stores the contents of the Y register at the specified memory location.

N	V	M	X	D	I	Z	C
.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Absolute	STY Loc	8C	3	5 ¹
Direct	STY DLoc	84	2	4 ^{1,2}
Direct indexed with X	STY DLoc,X	94	2	5 ^{1,2}

¹Subtract 1 cycle if X + 1 (8-bit index registers).²Add 1 cycle if direct register low (DL) is not equal to 0.**STZ***Store Zero*

Operation: Stores 0 at the specified memory location.

N	V	M	X	D	I	Z	C
.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Absolute	STZ Loc	9C	3	5 ¹
Absolute indexed with X	STZ Loc,X	9E	3	6 ¹
Direct	STZ DLoc	64	2	4 ^{1,2}
Direct indexed with X	STZ DLoc,X	74	2	5 ^{1,2}

¹Subtract 1 cycle if M = 1 (8-bit memory/accumulator).²Add 1 cycle if direct register low (DL) is not equal to 0.³Add 1 cycle if adding index crosses a page boundary.

SWA — See XBA

TAD — See TCD

TAS — See TCS

TAX*Transfer Accumulator to X Register*

Operation: Copies the contents of A into X.

N	V	M	X	D	I	Z	C
*	*	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Implied	TAX	AA	1	2

TAY*Transfer Accumulator to Y Register*

Operation: Copies the contents of A into Y.

N	V	M	X	D	I	Z	C
*	*	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Implied	TAY	A8	1	2

TCD (or TAD)*Transfer C Accumulator to Direct Register*

Operation: Copies the contents of C (the 16-bit accumulator) into D.

N	V	M	X	D	I	Z	C
*	*	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Implied	TCD	5B	1	2

430 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

TCS (or TAS)

Transfer C Accumulator to Stack Pointer

Operation: Copies the contents of C (the 16-bit accumulator) into S.

N V M X D I Z C

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Implied	TCS	1B	1	2

TDA — See TDC

TDC (or TDA)

Transfer Directo Register to C Accumulator

Operation: Copies the contents of D into C, the 16-bit accumulator.

N V M X D I Z C
 * *

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Implied	TDC	7B	1	2

TRB

Test and Reset Bits

Operation: ANDs the memory operand with the complement of the accumulator. Each 1 bit in the accumulator becomes 0 in the operand; each 0 bit in the accumulator leaves the corresponding operand bit unchanged.

N V M X D I Z C
 *

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Absolute	TRB Loc	1C	3	5 ¹
Direct	TRB DLoc	14	2	4 ^{1,2}

¹Subtract 1 cycle if M = 1 (8-bit memory/accumulator).

²Add 1 cycle if direct register low (DL) is not equal to 0.

TSA — See TSC

TSB*Test and Reset Bits*

Operation: ORs the memory operand with the accumulator. Each 1 bit in the accumulator becomes 1 in the operand; each 0 bit in the accumulator leaves the corresponding operand bit unchanged.

N	V	M	X	D	I	Z	C
.	*	.

Addressing Mode	Assembler Format		Opcode	Bytes	Cycles
Absolute	TSB	Loc	0C	3	5 ¹
Direct	TSB	DLoc	04	2	4 ^{1,2}

¹Subtract 1 cycle with M = 1 (8-bit memory/accumulator).

²Add 1 cycle if direct register low (DL) is not equal to 0.

TSC (or TSA)*Transfer Stack Pointer to C Accumulator*

Operation: Copies the contents of S into C, the 16-bit accumulator.

N	V	M	X	D	I	Z	C
*	*	.

Addressing Mode	Assembler Format		Opcode	Bytes	Cycles
Implied	TSC		3B	1	2

TSX*Transfer Stack Pointer to X Register*

Operation: Copies the contents of S into X.

N	V	M	X	D	I	Z	C
*	*	.

Addressing Mode	Assembler Format		Opcode	Bytes	Cycles
Implied	TSX		BA	1	2

432 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

TXA

Transfer X Register to Accumulator

Operation: Copies the contents of X into A.

N	V	M	X	D	I	Z	C
*	*	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Implied	TXA	8A	1	2

TXS

Transfer X Register to Stack Pointer

Operation: Copies the contents of X into S.

N	V	M	X	D	I	Z	C
.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Implied	TXS	9A	1	2

TXY

Transfer X Register to Y Register

Operation: Copies the contents of X into Y.

N	V	M	X	D	I	Z	C
*	*	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Implied	TXY	9B	1	2

TYA

Transfer Y Register to Accumulator

Operation: Copies the contents of Y into A.

N	V	M	X	D	I	Z	C
*	*	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Implied	TYA	98	1	2

TYX*Transfer Y Register to X Register*

Operation: Copies the contents of Y into X.

N	V	M	X	D	I	Z	C
*	*	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Implied	TYX	BB	1	2

WAI*Wait for Interrupt*

Operation: Puts the processor into a low-power idle state until it receives an external hardware interrupt.

N	V	M	X	D	I	Z	C
.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Implied	WAI	CB	1	3

XBA*Exchange B and A Bytes of Accumulator***SWA***Swap Accumulator Bytes*

Operation: Swaps B (high byte) and A (low byte).

N	V	M	X	D	I	Z	C
*	*	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Implied	XBA	EB	1	3

XCE*Exchange Carry and Emulation Bits*

Operation: Swaps the processor status register's C and E bits. If C is 1, the processor enters the 8-bit emulation mode; if C is 0, it enters the full 16-bit native mode.

N	V	M	X	D	I	Z	C
.	E

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Implied	XCE	FB	1	2

APPENDIX D

Requirements for Using Tool Sets

With the exception of the Tool Locator, each tool set requires other tool sets to be active. Table D-1 shows which tool sets you must start to use a particular tool set. The tool sets are listed in the order in which you must start them.

Shut down the tool sets in the following order:

- Desk Manager
- Font Manager, Print Manager, and other tool sets not listed here
- Menu Manager
- Window Manager
- Control Manager
- Dialog Manager
- Event Manager
- LineEdit
- QuickDraw
- Miscellaneous Tools
- Memory Manager (after freeing all allocated memory)
- Tool Locator

Loading RAM-Based Tools from Disk

Before starting the Window Manager, you must call LoadTools to read any needed RAM-based tools from disk. This includes not only the tool sets that are entirely on disk (e.g., the Window, Menu, and Control Managers), but also additional tools for some of the tool sets in ROM, such as QuickDraw's oval-drawing routines. If your program needs these tools, it must load them into memory. LoadTools requires a list of the identification numbers and minimum version numbers for the tool sets you want it to load from disk. Table D-2 lists the tool set numbers.

Table D-1

To use ↓		You must start the following tool sets:									
	Tool Locator	Memory Manager	Misc. Tools	QuickDraw	Event Manager	Window Manager	Control Manager	LineEdit	Dialog Manager	Menu Manager	Desk Manager
Tool Locator	•										
Memory Manager	•	•									
Misc. Tools	•	•	•								
QuickDraw	•	•	•	•							
Event Manager	•	•	•	•	•						•
Window Manager*	•	•	•	•	•	•	•				
Control Manager	•	•	•	•	•	•	•				
LineEdit	•	•		•		•		•			
Dialog Manager	•	•	•	•	•	•	•	•	•	•	
Menu Manager	•	•	•	•	•	•	•			•	
Desk Manager	•	•	•	•		•	•	•	•	•	•

*The Control Manager and Menu Manager are only required if you want to use the Window Manager's TaskMaster.

Table D-2

ROM-Based Tools Sets	
1	Tool Locator
2	Memory Manager
3	Miscellaneous Tools
4	QuickDraw II
5	Desk Manager
6	Event Manager
7	Scheduler
8	Sound Manager
9	Apple Desktop Bus
10	SANE
11	Integer Math Tools
12	Text Tools
13	(Used internally)

RAM-Based Tools Sets	
14	Window Manager
15	Menu Manager
16	Control Manager
17	Loader
18	Quickdraw Auxiliary Routines
19	Print Manager
20	LineEdit
21	Dialog Manager
22	Scrap Manager
23	Standard File Operations
24	Disk Utilities
25	Note Synthesizer
26	Note Sequencer
27	Font Manager
28	List Manager

Working Space for Tool Sets

Many tool sets require working space in bank 0, and you must make the Memory Manager allocate it with a `NewHandle` call. Specifically, QuickDraw requires three pages (or \$300 bytes), while the following sets each require one page (\$100 bytes):

Event Manager (shares its page with the Window Manager)

Control Manager (shares its page with the Dialog Manager)

Menu Manager

LineEdit

Font Manager

Print Manager

Sound Manager

Standard File Operations

SANE

Index

; (comment specifier), 35
\$ (hexadecimal specifier), 34-35
= (ProDOS wildcard character), 154
% (binary specifier), 34-35
(APW system prompt), 47
&LAB (symbolic label in macro definition), 134
___ (underscore prefix for tool calls), 166

65816 instruction set, 90-131,
 401-433
 list of, 91-93
 See also *Quick Index*
65816 internal registers, 13-19
65816 operand addressing modes,
 summary of, 74

A

A register (Accumulator), 15
Absolute addressing modes, 75-76
Absolute indexed addressing modes,
 78
Absolute indexed indirect addressing
 mode, 79-80

Absolute indirect addressing mode,
 77-78
Absolute long indexed with X
 addressing mode, 79
Accumulator (A) register, 15
Accumulator addressing mode, 75
Activate (window) event, 274
Active controls, 359
Active window, 299
Add and subtract macros, 146,
 150
Adding binary numbers, 4
Addition, 100-101
Address bus, 10
Addressing modes, operand formats
 for, 86
Addressing modes, summary tables
 of, 74, 86-87
AINCLUDE subdirectory on APW
 disk, 143
Alert
 box, 362, 364-367
 stages of an, 366
 template, 395-396
 window, 294-295
Alternate mnemonics for 65816
 instructions, 91, 93

Animated pixel image program
 (ANIMATE), 254-255
 Animation, 249, 253-254
 Apple II graphics modes, 191-192
 Apple IIGS
 internal architecture of, 19-22
 memory in, 22-25
 Monitor, 25-26
 Programmer's Workshop macros,
 143-154
 Programmer's Workshop,
 starting, 47
 Programmer's Workshop
 commands, 54-58
 super high-resolution graphics
 modes, 192
 Toolbox, 26, 155-190
 Apple menu, 327, 331, 336
 Application program, general
 structure of a, 168
 APW — See *Apple IIGS
 Programmer's Workshop*
 Arcs, drawing, 223
 Arithmetic instructions, 99-105
 Arithmetic operators, 43, 45
 Arithmetic, signed, 104
 Arranging two numbers in increasing
 order, 110
 ASCII character table, 400
 ASM65816 command, 54
 ASML (assemble and link)
 command, 55
 ASMLG (assemble, link, and go)
 command, 55
 ASSEMBLE command, 55
 Assembler, 27
 Assembler directives, 35-43
 See also *Quick Index*

Assembler directives, summary
 tables of, 37, 43
 Assembly language program, steps
 in developing an, 28-31
 Attributes, pen, 198
 Auto-key event, 274
 Automating the assembly process,
 62

B

B (Break command) flag, 19
 Banks \$E0 and \$E1 in Apple IIGS,
 23-24
 Banks, memory, 11
 BEEP (speaker-beeping program),
 48-49
 Binary (base 2) numbering, 1-7
 Binary digits (bits), 2
 Binary numbers, adding, 4
 Binary-coded decimal (BCD)
 numbers, 99-101, 103
 Bit (binary digit), 2
 Bit manipulation instructions,
 119-123
 Bit maps, Macintosh, 238-240
 Bit-testing instructions, 122-123
 Block move addressing mode, 85
 Block move instructions, 96-99
 Blocks in memory, 156
 Borrow in subtraction, 103
 BoundsRect, QuickDraw, 242, 245
 Bowtie-drawing program
 (BOWTIE), 225, 227
 Branching directives, 140-141
 Break command (B) flag, 19
 Breakpoints, 71-72

Breakpoints subdisplay, debugger,
66
BRK (breakpoint flag) on debugger
screen, 65
BUILD files, 62
Button-reading tool call, 286
Buttons, 159, 357
Byte (8-bit value), 4
Byte Works, Inc., The, 27
Bytes, packed, of BCD digits,
100

C

Calculation calls for shapes, 225-226
Calling conventions for tools,
166-168
Carry (C) flag, 17-18
CATALOG command, 54
Caution Alert box, 365
CDAs — See Classic desk
accessories
CHANGE command, 54
Check boxes, 159, 358
Check error macro, 147, 154
Circles, drawing, 224
Classic desk accessories (CDAs),
162
ClipRgn (clip region), Quickdraw,
246
Clock speeds, 12
Close box (go-away region),
295-296, 302, 311
CMOS, 10
Code segment, 36
Codes, event, 277-279

Color tables, 205, 266-269
Color-dithering program
(DITHERS), 232-233
Colors, 202, 205-207
320 mode, 205-206
640 mode, 206-208
Comment field, 35
Common programming errors,
189-190
Compare instructions, 105, 109-111
using branch instructions with,
109-111
Conceptual drawing space,
Quickdraw, 192-194
Conditional transfer (branch)
instructions, 107-111
Content region of a window, 295
Control Manager, 159-160, 356
Control Panel, 162, 275
Control transfer instructions,
106-111
Controls, on-screen, 159-160
Controls, predefined, 357-359
Conversion commands, debugger,
69, 71
Converting decimal values to binary,
3
Converting hexadecimal numbers to
decimal, 399
Copy (pen mode), 198
COPY file command, 56
Copying between programs, 186
Correcting typing errors, 50-53
CREATE subdirectory command,
56
Creating macro libraries, 142-143
Crystal, 12

D

- D (decimal mode) flag, 19
- D (Direct) register, 16
- Data Bank Register (DBR), 15
- Data bus, 10
- Data formats for numeric values, 99-100
- Data transfer instructions, 94-99
- Date and time operations, 254, 259-266
- DEBUG command, 57
- DebugD (direct page on debugger screen), 65
- Debugger, 29-31, 63-72
 - commands, summary table of, 67-69
 - leaving the, 72
 - starting the, 63-64
- Decimal mode (D) flag, 19
- Decimal numbers, 99-100
- Decimal values, converting to binary, 3
- Decimal values, converting to hexadecimal, 399
- Decision sequence, three-way, 110-111
- Decrement instructions, 104-105
- Default color table — See *Standard color table*
- Define macros, 146, 150-151
- DELAY subroutine, 262, 264, 266
- Delays, generating, 261-266
- DELETE file command, 56
- Desk accessories, 161-162
- Desk accessory event, 275
- Desk Manager, 161-162, 275
- Desk scrap, 162
- Dialog box, 161, 362-364, 366-367
- Dialog Manager, 161, 362, 367-372
 - item lists, 367-372
 - item types, 368-372
- Dials, 358
- Digital Oscillator Chip (DOC), 162
- Direct addressing mode, 80-81
- Direct indexed addressing modes, 82
- Direct indexed indirect addressing mode, 83
- Direct indirect addressing modes, 81-82
- Direct indirect indexed addressing modes, 82-83
- Direct page, 11
- Direct register, 16
- Directives, assembler, 35-42
 - summary table of, 37, 43
 - See also *Quick Index*
- Disabled menu, 329
- Disassembly commands, debugger, 69-70
- Disassembly subdisplay, debugger, 66
- Displaying a pixel image, 247-249
- Displaying text, 232, 236-237
- Dithering in 640 mode, 207-209, 232
- DITHERS (color-dithering program), 232-233
- DOC (Digital Oscillator Chip), 162
- Document window, 294-295
- Double high- and low-resolution graphics modes, 192
- Drawing
 - arcs, 223
 - circles, 224

- lines, 210-211
- ovals, 223-224
- polygons, 218-223
- rectangles, 211-218
- regions, 224-225
- Drawing space, Quickdraw
 - conceptual, 192-194

E

- Edit (selection in menu), 327
- EDIT command, 54
- Editing commands, debugger, 67, 69
- Editor, 29, 50-54
 - leaving the, 53-54
 - starting the, 50
- Editor commands, 51-52
- Effective address calculations, 86
- Elapsed time in event record, 279
- Emulation mode, 13, 17, 19,
 - 127-128
 - status bits in, 17, 19
 - switching to, 127-128
- ENABLE command, 57
- Enabled menu, 328
- Ensoniq Digital Oscillator Chip (DOC), 162
- Equate directives, 40-41
- Errors, correcting, 50-53
- Errors, programming, common, 189-190
- Event
 - codes, 277-279
 - definition of, 160, 271
 - loop, 272-273
 - mask, 283-284

- message, 279
- priorities, 276-277
- queue, 160, 271, 276-277
- record, 271, 277-281
- types, 273-276
- Event Manager, 160-161, 271, 281-286
- EXE (shell load) files, 62
- EXEC command, 54
- EXEC files, 62

F

- Fast Processor Interface (FPI) chip, 20
- File (selection in menu), 327
- File control directives, 39
- FILETYPE command, 57, 62
- Firmware, 20
- Fixed blocks in memory, 156
- Flags, status, 17
- Flowchart, 28
- Font Manager, 162, 359
- Format disk (INIT) command, 57
- Frame (rectangle boundary), 211

G

- General-purpose registers, 15
- Generating delays, 261-266
- Global labels, 38
- Go-away region (close box), 310
- GrafPort, Quickdraw, 242-247
 - record, 244
- Graphics modes, 191-192
- Grow box, 296-297, 302

H

Handle (pointer to a pointer), 157, 169-171
 HELP command, 54
 Hertz (cycles per second), 12
 Hexadecimal (base 16) numbering, 1, 7-8
 Hexadecimal to decimal conversion, 399
 High-resolution graphics mode, 191

instructions, 128-131
 request, 13
 requests, disabling, 19
 service routine (handler), 128
 vectors, 13, 128
 software, 130
 IRQ disable (I) flag, 19
 Item
 lists, 367-372
 template, 395-396
 types, 368-372

I

Immediate addressing mode, 74-75
 Implied addressing mode, 75
 Inactive controls, 360
 Inactive window, 299
 Increment and decrement
 instructions, 104-105
 Index register select bit (X), 19
 Index registers (X and Y), 15
 Information bar, 298-299
 INIT (format disk) command, 57
 Instruction set, 65816, 90-131, 401-433
 See also *Quick Index*
 Instruction set, 65816, list of, 91-93
 Integer Math Tools, 163
 Integers, 168
 Integrated Woz Machine (IWM), 22
 Internal registers, 65816, 13-19
 Interrupt(s)
 control instructions, 129
 defined, 12-13
 disable (I) flag, 19

K

K (abbreviation for 1024), 4
 K/PC on debugger screen, 65
 KEEP option for ASML command, 184
 KEEP option for LINK command, 56
 KEY (keystroke modifier) on
 debugger screen, 64
 Key commands for menu items, 334
 Key-down event, 274
 Keyboard events, 274, 338-340

L

L (language card bank) on debugger
 screen, 65
 Label field, 33
 Labels
 characters in, 33
 global, 38
 local, 38
 Language card bank (L on debugger
 screen), 65

Line Editor (LineEdit), 163, 366
 Lines, drawing, 210-211
 LINK command, 56
 Linker, 29
 List Manager, 162
 Listing, columns in a, 60-61
 Listing directives, 41-42, 141
 Load and store instructions, 94-96
 Load and store macros, 146, 149
 Loading RAM-based tools from disk, 434
 Local labels, 38
 LocInfo data structure, 243-244
 Logical instructions, 119-122
 Logical operators, 45-46
 Longword (4-byte value), 168
 Low-resolution graphics mode, 191

M

M (machine state register) on debugger screen, 65
 M (memory/accumulator select bit), 19
 MACGEN (Macro Library Generator) utility, 142-143, 154
 Macintosh bit maps, 238-240
 Macro, defined, 133
 Macro directives, 136-141
 Macro language directives, 136, 138
 Macro libraries, creating, 142-143
 Macro libraries, updating, 143
 Macro library directives, 139
 Macros
 compared with subroutines, 133
 contents of, 134-135

 on the Programmer's Workshop Disk, 143-154
 using predefined, 154
 Managers in Apple IIGS Toolbox, 158-162
 Mask, pen, 201-202
 Master color value, 205
 Master pointer, 157
 Master scan line control byte, 196-197, 267
 Mega II chip, 20
 Megabyte, 10
 Memory
 banks, 11
 commands, debugger, 68-70
 in Apple IIGS, 22-25
 map, Apple IIGS, 22, 24
 organization, 11-12
 protection subdisplay, debugger, 68
 shadowing, 23
 Memory Manager, 156-157, 169
 starting the, 169
 Memory/accumulator select bit (M), 19
 Mensch, William D., Jr., 131
 Menu
 bar, 326-329
 events, 337-340
 items, 329-331
 modifiers, 334-337
 program (MENUS), 340-341
 Menu Manager, 158, 326
 Menu/item line list, 332-334
 Message, event, 279
 Messages, displaying, 232, 236-237
 Microsecond, 12

- Minimum version numbers of tool sets, 172-173
 - Minipalettes (640 mode color table groups), 206
 - Miscellaneous Tools tool set, 164, 169, 259-261
 - starting, 169
 - Mnemonic (opcode) field, 34
 - Mnemonics for 65816 instructions, 91-93
 - Modal
 - dialog boxes, 363
 - dialog box program (MODALD), 378
 - events, 375
 - programs, 271-272
 - Mode
 - control bits, 17
 - control instructions, 127-128
 - directives, 42
 - macros, 147, 153
 - Model program, listing for, 178-182
 - Modeless dialog boxes, 363-364
 - Modeless events, 376
 - Modifiers in event record, 279-281
 - Modifiers, menu, 334-337
 - Monitor, Apple IIGS, 25-26
 - Mouse events, 273, 337-338
 - Mouse pointer location, 279
 - Mouse pointer tool call
 - (ShowCursor), 232
 - Mouse-down event, 273
 - Mouse-reading tool calls, 286
 - Mouse-up event, 273
 - MouseWrite word processor, 29
 - MOVE file command, 57
 - Move macros, 146, 151-152
 - Multiprecision numbers, adding, 101
 - Multiprecision numbers,
 - subtracting, 103
 - Multisegment programs, 63
- N**
- N (negative) flag, 19
 - Nanosecond, 12
 - Native mode, 13, 17-19, 127
 - status bits in, 17-19
 - switching to, 127
 - NDA's — See *New desk accessories*
 - Negative (N) flag, 19
 - New desk accessories (NDAs), 162, 340
 - Nibble (4-bit number), 199
 - Note alert box, 365
 - Note Synthesizer, 162
 - Null event, 275-276
 - Numbers
 - arranging in increasing order, 110
 - binary-coded decimal (BCD), 99-101, 103
 - shifting signed, 126
 - signed, 99, 104
 - unsigned, 99
 - Numeric values, data formats for, 99-100
- O**
- Object module, 29
 - Opcode (mnemonic) field, 34-35
 - Operand addressing modes,
 - summary of, 74

Operand field, 34-35
 Operand formats for addressing modes, 86
 Operating modes, 13
 Operators, 42-47
 ORCA/M assembler, 27
 Ovals, drawing, 223-224
 Overflow, 19

P

P (Processor Status Register), 17-19
 Packed bytes of BCD digits, 100
 Page (256-byte memory unit), 11
 Page, direct, 11
 Page, zero (in 6502), 11
 Pattern, pen, 198-200
 PBR (Program Bank Register), 16
 PC (Program Counter) register, 16
 Pen, QuickDraw, 198-203
 attributes, 198
 location (PenLoc), 198
 mask, 201-202
 mode (PenMode), 198
 pattern (PenPat), 198-200
 size (PenSize), 198
 state, 198, 202-203
 Pencil display program (PENCIL), 249-250
 Pixel (picture element), 194-195
 Pixel image, 238-242, 247-249, 254
 Point in Quickdraw's drawing space, 194-195
 Pointer (4-byte address), 156, 168
 POLY (triangle-drawing program), 219-220

Polygons, drawing, 218-223
 PortRect, Quickdraw, 245
 Predefined controls, 357-359
 PREFIX command, 57
 PrepareToDie subroutine, 169
 Print Manager, 162
 Printer interface cards, 59-60
 Priorities, event, 276-277
 Processor status bit instructions, 123
 Processor Status Register (P), 17-19
 ProDOS 16 macros, 144-145
 Program Bank Register (PBR), 16
 Program control directives, 36-38
 Program Counter (PC), 16
 Program counter relative addressing modes, 83-84
 Program identification number, 169
 Program model, listing for, 178-182
 Programmer's Workshop —
 See Apple II GS Programmer's Workshop
 Programming errors, common, 189-190
 Programs, copying between, 186
 Pull-down menus, 328
 Push and pull instructions, 115-119
 Push and pull macros, 144-149

Q

Q (quagmire register) on debugger screen, 65
 Queue, event, 160, 271, 276-277
 QuickDraw II
 conceptual drawing space, 192-194
 described, 157-158

drawing environment, 192-196
starting and stopping, 196-197

R

Radio buttons, 159, 358
RAM subdisplay, debugger, 65-66
RAM-based tools, 171-176, 434
loading from disk, 434
Read-Modify-Write instructions, 86-88
Rectangle(s)
drawing program (RECTS), 214
drawing, 211-218
frame (boundary) of, 211
round-cornered (roundrects), 223
Regions, drawing, 224-225
Register commands, debugger, 68-70
Register subdisplay, debugger, 64-65
Register transfer instructions, 96
Registers, 65816, 13-19
Relational operators, 46-47
Relocatable blocks in memory, 156
RENAME file command, 57
Repeat count for DC directives, 40
Request, interrupt, 13
Results, reserving stack space for, 167
ROM, Apple IIGS, 25
ROM-based tool sets, 171
Rotate instructions, 125
Roundrects (round-cornered rectangles), 223

S

S register (Stack Pointer), 15-16
S16 (ProDOS 16 system load) files, 62
SANE (Standard Apple Numerics Environment), 163
Scan line control byte (SCB), 196-197, 266-267
Scrap Manager, 162
ScreenMode global equate, 197
Scroll bars, 159, 297-298, 302, 358-359
Serial Communications Controller (SCC), 22
Shadowing, memory, 23
Shapes, animating, 254
Shapes, calculation calls for, 225-226
Shell load (EXE) file, 29
Shift and rotate instructions, 124-126
Shift macros, 146, 152-153
Shifting signed numbers, 126
SHOW command, 57
SHOWTEXT (text display program), 286-287
Shutting down tool sets, 176-177, 188-189, 434
Sign bit, 5
Signed arithmetic, 104
Signed numbers, 5, 99, 104, 126
shifting, 126
Single-step commands, debugger, 66-67
Software interrupts, 130
Sound Manager, 162

- Source program, 27
 - Source statements, 32
 - Space allocation directives, 39-40
 - Speaker-beeping program (BEEP), 48-49
 - Stack
 - addressing modes, 84
 - described, 15-16
 - instructions, 113, 115-119
 - space, reserving for a result, 167
 - subdisplay, debugger, 65
 - Stack Pointer (S), 15-16
 - Stack relative addressing modes, 84-85
 - Stages of an alert, 366
 - Standard color table
 - 320 mode, 205
 - 640 mode, 206
 - Standard File Operations tool set, 163
 - Start-up sequence for tool sets, 186
 - Starting ROM-based tool sets, 171
 - Status bit instructions, 123
 - Status Register, Processor, 17-19
 - flags in, 17
 - Stop alert box, 365
 - Store instructions, 96
 - Store macros, 146, 149
 - STR (string) macro, 150
 - Subdisplays on the debugger screen, 64-66
 - Subroutine instructions, 111-113
 - Subroutines compared with macros, 133
 - Subtraction, 103
 - Subtraction macros, 146, 149
 - Super high-resolution graphics modes, Apple IIGS, 192
 - Switch (applications) events, 275
 - Switching between native and emulation mode, 127-128
 - Symbolic parameter directives, 139-140
 - System load (S16) file, 29
 - System menu bar, 327
- T**
- Task
 - code, 309-310
 - mask, 308-309
 - record (TaskMaster event record), 308-309
 - TaskMaster, 307-311
 - Text display program (SHOWTEXT), 286-287
 - Text, displaying, 232, 236-237
 - Text Tools tool set, 163
 - Three-way decision sequence, 110-111
 - Tick, 279
 - Time and date operations, 254, 259-266
 - Title bar in a window, 295, 302
 - Tool calls, making, 166-168
 - Tool calls, using in programs, 165-166
 - Tool Locator, 156, 169
 - starting the, 169
 - Tool set interactions, 164-165, 435
 - Tool set numbers, 172, 436

Tool sets

- minimum version numbers of, 172-173
- shutting down, 176-177, 188-189, 434
- start-up sequence for, 186
- working space for, 169-171, 436-437

Tool table, 171

Toolbox — See *Apple IIGS Toolbox*

Top-down program design, 31-32

Trace commands, debugger, 66-67

Triangle-drawing program (POLY), 219-220

Two's-complement numbers, 6-7

TYPE command, 55

U

Unconditional transfer instructions, 106-107

Underscore prefix for tool calls, 166

Unsigned numbers, 99

Update (window) event, 274-275

Updating macro libraries, 143

User-defined events, 275

Utility macros, 144-154

V

V (overflow) flag, 19

Vectors, interrupt, 13, 128

Video Generator Chip (VGC), 22

W

Weights of bit positions, 2-3

Western Design Center, 10

Window

alert, 294, 295

components, 295-299

content region of a, 295

definition of, 294

display program (WINDOWS), 311-313

document, 294-295

events, 274-275

Window Manager, 158, 294

WINDOWS (window display program), 311-313

Word (2-byte value), 168

Working space for tool sets, 169-171, 436-437

Write macros, 147, 153-154

X

X (index register select) bit, 19

X register, 15

Y

Y register, 15

Z

Zero (Z) flag, 18

Zero page (in 6502), 11

Zoom box, 296, 302

Quick Index

Instructions

ADC, 100-101, 402
AND, 119-120, 403
ASL, 124, 403
BCC, 106, 108, 404
BCS, 106, 108, 404
BEQ, 106, 108, 404
BGE, 108, 404
BGE, 93, 404
BIT, 120, 122, 405
BLT, 93, 108, 404
BMI, 106, 108, 405
BNE, 106, 108, 406
BPL, 106, 108, 406
BRA, 106-107, 406
BRK, 130, 407
BRL, 106-107, 407
BVC, 106, 108, 407
BVS, 106, 108, 408
CLC, 100-101, 408
CLD, 100, 408
CLI, 129-130, 408
CLV, 100, 104, 409
CMA, 93, 409
CMP, 100, 105, 109-111, 409
COP, 130, 410
CPX, 100, 105, 109-111, 410
CPY, 100, 105, 109-111, 410

DEA, 93, 411
DEC, 100, 104, 411
DEX, 100, 104, 411
DEY, 100, 104, 411
EOR, 120-121, 412
INA, 93, 412
INC, 100, 104, 412
INX, 100, 104, 413
INY, 100, 104, 413
JML, 106-107, 413
JMP, 106, 414
JSL, 111, 113, 414
JSR, 111-112, 414
LDA, 94-95, 415
LDX, 94-95, 415
LDY, 94-95, 416
LSR, 124, 416
MVN, 95-99, 417
MVP, 95-99, 417
NOP, 131, 417
ORA, 120-121, 418
PEA, 116, 118, 418
PEI, 116, 118, 419
PER, 116, 119, 419
PHA, 115-117, 419
PHB, 115-116, 419
PHD, 115-116, 420
PHK, 115-116, 420
PHP, 115-116, 420

PHX, 115-116, 420
 PHY, 115-116, 421
 PLA, 115-116, 421
 PLB, 115-116, 421
 PLD, 115-116, 421
 PLP, 115-116, 422
 PLX, 115-116, 422
 PLY, 115-116, 422
 REP, 120, 123, 422
 ROL, 124-125, 423
 ROR, 124-125, 423
 RTI, 129-130, 424
 RTL, 111, 113, 424
 RTS, 111-112, 424
 SBC, 100, 103, 425
 SEC, 100, 103, 425
 SED, 100-101, 103, 426
 SEI, 129-130, 426
 SEP, 120, 123, 426
 STA, 95-96, 427
 STP, 131, 427
 STX, 95-96, 428
 STY, 95-96, 428
 STZ, 95-96, 428
 SWA, 93, 433
 TAD, 93, 429
 TAS, 93, 430
 TAX, 95-97, 429
 TAY, 95-97, 429
 TCD, 95-97, 429
 TCS, 95-97, 430
 TDA, 93, 430
 TDC, 95-97, 430
 TRB, 120, 122, 430
 TSA, 93, 431
 TSB, 120, 122, 431
 TSX, 95-97, 431

TXA, 95-97, 432
 TXS, 95-97, 432
 TXY, 95-97, 432
 TYA, 95-97, 432
 TYX, 95-97, 433
 WAI, 130-131, 433
 WDM, 131
 XBA, 95-97, 433
 XCE, 127, 433

Assembler Directives

65816, 37, 42
 65C02, 37, 42
 ABSADDR, 37, 41
 ALIGN, 43
 APPEND, 37, 39
 CASE, 43
 COPY, 37, 39
 DATA, 37-38
 DC, 37, 39
 DS, 37, 39
 EJECT, 37, 42
 END, 36-37
 ENTRY, 37-38
 EQU, 37, 40
 EXPAND, 43
 GEN, 138, 141
 GEQU, 37, 40
 IEEE, 43
 INSTIME, 43
 KEEP, 37, 39
 LIST, 37, 41
 LONGA, 37-38
 LONGI, 37-38
 MACRO, 134, 137

MCOPY, 137, 139, 165
MDROP, 137, 139
MEM, 43
MEND, 135, 137
MERR, 43
MLOAD, 137, 139
MSB, 43
OBJCASE, 43
ORG, 37, 40
PRINTER, 37, 41
PRIVATE, 43
PRIVDATA, 43
RENAME, 43
SETCOM, 43
START, 36-37
SYMBOL, 37, 41
TITLE, 37, 42
TRACE, 139, 141
USING, 37-38

Tool Calls

Alert, 393-394
Button, 282, 286
CautionAlert, 365, 394-396
ClearScreen, 210
CloseDialog, 374-375
ClosePoly, 218-219
CloseRgn, 225-226
CloseWindow, 301, 307
CopyRgn, 226
CtlShutDown, 312, 361
CtlStartup, 312, 361
DialogSelect, 375-376
DialogShutDown, 367
DialogStartup, 367

DisposeAll, 188
DisposeHandle, 188
DoKeyDown, 277
DoMouseDown, 277
DrawChar, 237
DrawCString, 237
DrawMenuBar, 333
DrawString, 237
EMShutDown, 281
EMStartup, 281-283
EraseOval, 224
ErasePoly, 219
EraseRect, 212
EraseRgn, 226
FillOval, 224
FillPoly, 219
FillRect, 212
FillRgn, 226
FixAppleMenu, 340
FixMenuBar, 333
FlushEvents, 282, 286
FrameOval, 224
FramePoly, 219
FrameRect, 211-212
FrameRgn, 226
GetBackPat, 210
GetColorTable, 267, 269
GetMouse, 282, 286
GetNextEvent, 281, 283-285, 307
GetPen, 203
GetPenMask, 204
GetPenPat, 204
GetPenSize, 204
GetPenState, 203
GetPort, 247
GetPortRect, 248
GetPrefix, 176

GetText, 375-377	PaintPoly, 219
GetTick, 266	PaintRect, 212
HideWindow, 302, 307	PaintRgn, 226
InitColorTable, 267, 269	PenNormal, 202-203
InsertMenu, 333	PenPat, 208
IsDialogEvent, 375-376	PPToPort, 247-249
LEShutDown, 367	QDShutDown, 197
LEStartup, 367	QDStartup, 196
Line, 210-211	Quit, 177
LineTo, 210-211	ReadAsciiTime, 259-260
LoadOneTool, 187, 225	ReadTimeHex, 259-260
LoadTools, 171, 187	Refresh, 301
MenuSelect, 337	RemoveDItem, 374
MenuShutDown, 333	ScrollRect, 248
MenuStartup, 333	SelectWindow, 302, 307
MMShutDown, 187	SetBackColor, 237
MMStartup, 169, 187	SetBackPat, 210
ModalDialog, 374-375	SetColorEntry, 267
Move, 203	SetColorTable, 267, 269
MoveTo, 203, 211	SetForeColor, 237
MTShutDown, 188, 260	SetMasterSCB, 267, 269
MTStartup, 169, 188, 260	SetOrigin, 248-249
NewDItem, 374	SetPenMask, 204
NewHandle, 169-170, 187	SetPenPat, 204
NewMenu, 333	SetPenSize, 204
NewModalDialog, 373	SetPenState, 203
NewModelessDialog, 374	SetPort, 247
NewRgn, 225-226	SetPortRect, 248-249
NewWindow, 300-306	SetPrefix, 176
NoteAlert, 365, 394-396	SetRect, 226
OffsetPoly, 226	SetRectRgn, 226
OffsetRect, 226	SetSolidBackPat, 208
OffsetRgn, 226	SetSolidPenPat, 208, 210
OpenPoly, 218-219	ShowCursor, 232
OpenRgn, 224, 226	ShowWindow, 301, 306
PaintOval, 224	SolidPattern, 210

StopAlert, 365, 394-396

SysFailMgr, 188

TaskMaster, 307-311

TLMountVolume, 176, 187

TLShutDown, 187

TLStartup, 169, 187

WindShutDown, 301

WindStartup, 301

Special Software Offer!

Get a running start on IIGS software development with all the sample programs from this book!

The software featured in Apple IIGS Assembly Language Programming is available for only \$9.95, plus \$3.00 for shipping and handling. Why spend several hours typing it in yourself? Send us your check, or call our 800 number, and we'll send you a 3.5" IIGS disk with all the source code, macros, exec files, and executable load files from the sample programs in the book.

This disk is not available in stores; it can only be purchased directly from the publisher.

Use this coupon to order or call 1-800 223-6834, ext. 479, and have your credit card information handy.

Mail to: Bantam Books, Inc. Dept. IIGS
666 Fifth Avenue
New York, NY 10103

Yes! Send me the Apple IIGS Assembly Language Program Disk (50049-X) for only \$12.95 (\$9.95 plus \$3.00 for UPS shipping and handling).

Name _____

Address _____

City _____ State _____ Zip _____

☐ My check or money order for \$12.95 is enclosed.
(Please make check payable to Bantam Books, Inc.)

☐ Charge my _____ Visa _____ MasterCard _____ American Express

Acct. # _____ Exp. Date _____

Signature _____

Please allow 3-4 weeks for delivery. Price shown in U.S. dollars. Price and availability subject to change without notice. This offer is not available in stores. No C.O.D.s.

IIGS—8/87

CHAPTER 6

The Apple IIGS Toolbox

In Chapter 2, I introduced the Apple IIGS Toolbox, the large collection of functionally-based *tool sets*. I also mentioned that tool sets are comprised of assembly language routines called *tools*. Some tool sets are built into ROM, others are entirely on the System Disk, and a few are divided between ROM and the System Disk.

To use tools that are on the System Disk, you must first read them into the computer's memory. Apple's manuals refer to such tools as *RAM-based tools*, to distinguish them from tools that are built into ROM. In time, if Apple adds more ROM to the IIGS, it's likely that all the tools will be stored in ROM. Until then, your program must have some way of knowing where to obtain the tools it needs.

Fortunately, the ROM includes a tool set called (appropriately) the *Tool Locator* that finds tools for you. You must use the Tool Locator in every program that includes tool calls, and that will be virtually every program you write for the IIGS. Two other tool sets that get involved in most programs are the Memory Manager and QuickDraw II, so I'll discuss all the tool sets briefly, but these three I'll describe in detail. After that, I will demonstrate how to initialize tool sets and how to access the tools within them. Finally, I will present some guidelines for using tools and provide a program "model" that includes the instructions, assembler directives, and tool calls that programs generally need.

Important: Like most software products, the Apple IIGS Programmer's Workshop will probably change from time to time. Apple may add, delete, and modify tools — or even tool sets — and thereby affect the accuracy of this book. For example, the name of a tool may change. Although I have no control over that, I intend to keep the book as up-to-date as possible. To do that, I ask for your help. If you encounter something that appears to be inaccurate in the book, please write to me in care of the publisher.

Tool Locator

When your program requests use of a tool (by issuing a tool “call”), the Tool Locator looks up the memory address of that tool and gives it to the 65816. Looking up the address takes two steps, and they involve numbers that the Apple IIGS designers have assigned to tool sets and the tools within them.

First, the Tool Locator uses the tool set number to look up the 4-byte address, or *pointer*, in a tool set address table. This pointer points to a second table, one that holds the addresses of the individual tools in the tool set. The Tool Locator uses the tool number to obtain from this second table the address of the routine for that tool — and that is the execution address it gives to the microprocessor.

A picture is worth 76 words here. Figure 6-1 shows how a tool call (SetPenMode, in this case) makes the Tool Locator look up the two pointers and eventually direct the 816 to the tool's routine in memory.

Memory Manager

The Memory Manager is a ROM-based tool set that controls the use of memory in the Apple IIGS; think of it as the computer's bookkeeper. When you load a program from disk, the System Loader calls the Memory Manager to request space for it. The Memory Manager, in turn, allocates a block of memory for the program and tells the System Loader the block's address.

A program always remains where the System Loader puts it; it is at a *fixed* location in memory. Not all blocks in memory are fixed, however; blocks that programs set up to hold data are often *relocatable*. The Memory Manager is free to move them if it needs that space for a program or another data block. For this reason, when your program requests a block of memory, the Memory Manager has a noteworthy way of reporting where that block is located.

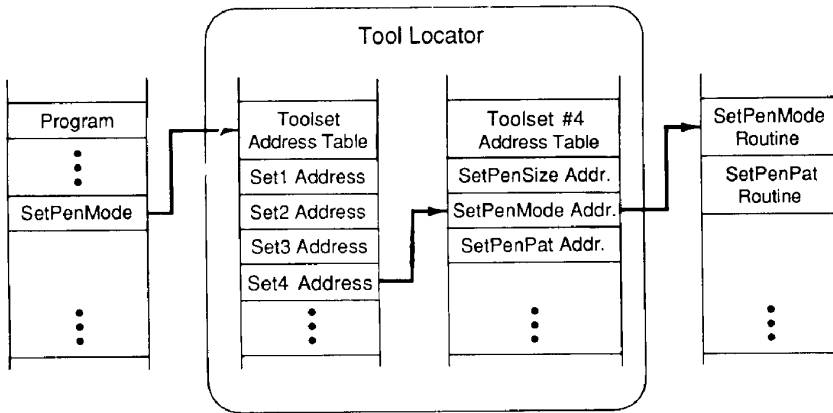


Figure 6-1

In response to your program's request, the Memory Manager allocates the block and puts its address in a 4-byte location called the *master pointer*. It does not give this address to the program, because it may move the block. Instead, the Memory Manager gives the program the address of the master pointer. This address is called the block's *handle*.

Note the logic behind this technique: the block may move (and its address change), but the address of the master pointer (i.e., the handle) never changes. The handle is, then, an identification number that the program can use at any time to access the allocated block, regardless of where it is in memory.

Even if your application needs no memory blocks for its own use, some of the Toolbox's tool sets require working space in memory bank 0, and your program must ask the Memory Manager to allocate it. (For example, QuickDraw II requires three pages, or 768 bytes). Finally, at the end of your program, you must ask the Memory Manager to free up or "deallocate" your memory block. To do this, you tell it to discard the block's handle.

QuickDraw II

Most, if not all, of your programs will involve displaying something on the screen. If the screen is in one of the regular Apple II graphics or text display modes, you could use Apple II emulation to produce your picture or text. However, to take advantage of the new super high-resolution mode (the

standard output for 65816 native-mode programming), you will need the services of the Apple IIGS's video display tool set, QuickDraw II.

QuickDraw II is responsible for every kind of display operation: drawing windows, graphics shapes (rectangles, ovals, arcs, etc.), menus, and text characters. Yes, even text! With super high-resolution, text is displayed as bit-mapped graphics, not as the built-in text characters the other display modes use. I'll describe QuickDraw II in more detail, and show you how to use it, in the next chapter.

Managers

In addition to the Tool Locator, Memory Manager, and QuickDraw II — which virtually all programs use — the Apple IIGS Toolbox contains a variety of other tool sets. Some have names ending with “Manager,” because they manage a certain type of operation.

Window Manager

The Window Manager lets you display windows. The simplest window is the blank full screen, but you can also create more complex windows that have a menu bar, scroll bars, and other components. The Window Manager also keeps overlapping windows under control.

The Window Manager relies on QuickDraw II to draw the elements in a window's border and the picture or text within it, and on the Control Manager (described shortly) to manage its components.

Menu Manager

The Menu Manager helps you create Macintosh-style menu bars and menus that you “pull down” from it using the mouse. Figure 6-2 shows an Edit menu that has been pulled down from a menu bar.

When you pull down a menu, the Menu Manager keeps track of which item you're pointing to. As soon as you release the mouse button, the Menu Manager reports the selection to the application program. The program, in turn, is responsible for executing whatever instructions are associated with that item.

The Menu Manager relies heavily on QuickDraw II's drawing and color capabilities. Since menus are part of a window, it also relies on the Window Manager.

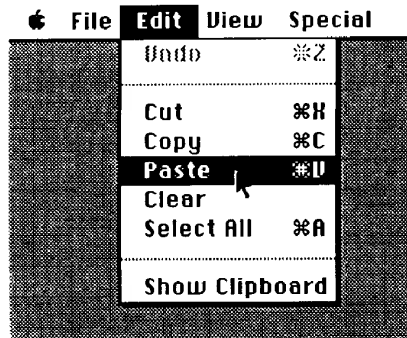


Figure 6-2

Control Manager

The Control Manager regulates on-screen *controls* — graphics objects that you activate with the mouse. The following controls are defined within the Control Manager:

- *Buttons* cause an immediate action when clicked or pressed with the mouse. They appear on the screen as round-cornered rectangles with a title centered inside.
- *Check boxes* act like on/off switches; they are checked with an X when on and empty (unchecked) when off. Check boxes are used to choose options from a list. They are generally used to specify some future action, instead of causing an immediate action of their own.
- *Radio buttons* are also on/off controls. But while any number of check boxes may be on, clicking a radio button on turns off all other radio buttons in that group. That is, they act like the station-selector buttons on a car radio.
- *Window scroll bars* are dials in which the moving part, called the “scroll bar” (or, sometimes, “thumb”), shows the position of the window relative to the total size of the document or picture being displayed.

Figure 6-3 shows the predefined controls, plus two other kinds of dials you

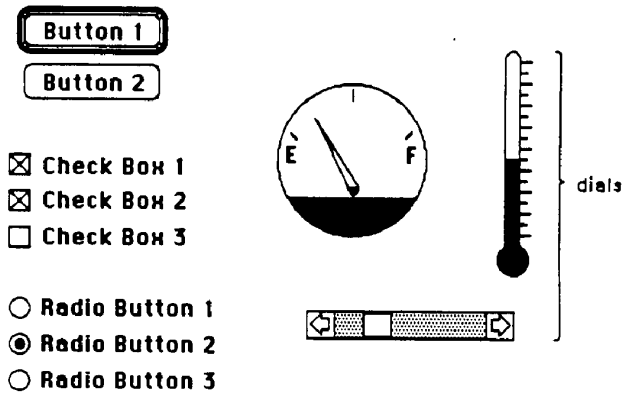


Figure 6-3

can define using the Control Manager. Since controls are graphics objects, the Control Manager works in conjunction with QuickDraw II; and because the controls are placed in a window, it also relies on the Window Manager.

Event Manager

Most programs interact with the user, letting him or her make selections or issue commands by pressing keys or clicking the mouse. The Apple IIGS manuals refer to these actions — pressing a key or clicking the mouse — as *events*. Some interactive programs sit and wait for an event from the user. This is common in educational programs, for example, where the program must wait for the user to answer a question. Other interactive programs, such as games, don't wait for events; they simply continue running, and deal with events as they occur.

The Event Manager keeps track of events by continually monitoring the keyboard and mouse. Every time the user presses a key or clicks the mouse button, the Event Manager takes note of what happened and records it in a chronological log called the *event queue*. (A queue is a stack that operates in “first-in/first-out” fashion, like a vending machine that dispenses candy. Compare this with the 65816's stack, which operates in “last-in/first-out” fashion, like a machine that holds trays in a cafeteria.)

The application program must, then, examine the event queue periodically to see if it contains any event records. If there are any records in the queue, the program must extract each one and do what it signifies (if anything), then continue processing records until the queue is empty.

To draw an analogy, the Event Manager acts like a receptionist in an office. Since the boss (the application program) is always busy, the Event Manager answers telephone calls (key and mouse button presses) and notes them on “While You Were Out” forms (event records). Each form is put on the boss’s desk (the event queue), beneath any that are already there. From that point, it’s up to the boss to respond.

Dialog Manager

Commands in menus normally perform one operation. (For example, “Quit” normally terminates the program.) However, commands that need more information from the user must present a *dialog box* (a rectangle that may contain text, controls, and icons) to request that information. Figure 6-4 shows a dialog box that lets the user specify the page setup for a print operation. Dialog boxes are the responsibility of the Dialog Manager.

Desk Manager

The Desk Manager handles small, memory-resident programs that usually provide on-screen equivalents of desk accessories — calculators, calendars, clocks, notepads, and so on. Being in memory, a desk accessory program can be summoned while a main application program is running; that is, the

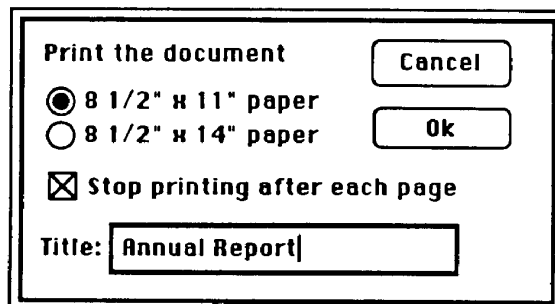


Figure 6-4

user can summon the desk accessory and do something with it, then continue the application as if nothing had happened. If you want to be able to use desk accessory programs while your application is running, you must activate the Desk Manager.

The Apple IIGS can handle two kinds of desk accessories, called (perhaps with a tip of the hat to Coca-Cola) *classic* and *new*. Classic desk accessories (CDAs) can be summoned from either 6502 emulation mode or 65816 native mode. To get a list of the installed CDAs, you press OpenApple-Control-Esc — the key combination that makes the Apple IIGS Control Panel available. Indeed, the Control Panel is itself a classic desk accessory. The Control Panel is built into ROM; other CDAs must be provided on disk.

New desk accessories (NDAs) can only be called from native mode. If the application program has provided for NDAs, their names appear automatically when the user selects the Apple menu on the menu bar at the top of the screen. With these names on the screen, you can activate any NDA by moving the mouse pointer to it and releasing the button. You may already be familiar with NDAs; they're the style of desk accessories available on the Macintosh.

Sound Manager

The *Sound Manager* uses the Ensoniq Digital Oscillator Chip (DOC) in the Apple IIGS to produce tones through the speaker. The DOC contains 32 individual oscillators, and you can combine them in pairs to produce up to 15 “voices.” A related tool set called the *Note Synthesizer* works with the Sound Manager and the DOC to create a polyphonic music synthesizer.

Other Managers

There are also a few other managers worth mentioning. The *Print Manager* lets your program print text or graphics without concern as to what kind of printer is connected to the computer.

The *Font Manager* lets you display text in any of several styles or “fonts.” It also lets you create fonts of your own design.

The *List Manager* lets you operate on lists of information, to search them, sort them, and so on.

The *Scrap Manager* provides the tools you need to move or copy text or graphics between programs (either two application programs, an application program and a desk accessory, or two desk accessories). The Scrap Manager gets its name from the fact that it keeps the data being transferred in a block of memory called the *desk scrap*.

Other Tool Sets

In addition to the managers, there are six other major tool sets: the Line Editor, Text Tools, Integer Math Tools, Standard File Operations, SANE, and miscellaneous tools.

Line Editor

The Line Editor (or *LineEdit*) provides basic line-editing capabilities. Among the things it lets you do are:

- Insert new text.
- Delete characters by backspacing over them.
- Select text to be cut or deleted using the arrow keys, or dragging through it with the mouse.
- Move or copy blocks of text.

Text Tools

There are two ways to display text on a text-mode screen. One way is to run your program in emulation mode in bank 0 and call the Monitor subroutines that output text. A better way is to use the Apple IIGS Text Tools. The Text Tools output text with the 65816 running in native mode in any bank.

Integer Math Tools

The Integer Math Tools include routines for operating on 2-byte and 4-byte signed integers, and on signed fixed-point numbers and their fractional parts. The operations the Integer Math Tools can do include multiplication, division, square root, sine, cosine, arc tangent, rounding, and conversions between data types.

Standard File Operations

The Standard File Operations tool set allows you to open and store files from within an application program. It can work with files on any drive in the system.

SANE

SANE, short for Standard Apple Numerics Environment, is the tool set for doing high-precision fixed-point and floating-point math operations. For

fixed-point computations, it uses a 64-bit data type. For floating-point, it can use any of three data types: single (32-bit), double (64-bit), or extended (80-bit). Besides rudimentary operations such as add, subtract, multiply, divide, and square root, SANE can do such jobs as:

- Compare numbers.
- Produce logs, exponentials, and trigonometric functions.
- Convert between binary and decimal or floating-point and integer.
- Calculate compound interest and annuity functions for financial institutions.
- Generate random numbers.

Miscellaneous Tools

With the tool calls in the miscellaneous tool set, you can:

- Access the battery backed-up RAM that keeps track of the date and time. This area of memory includes all the Control Panel parameters, including date and time, display mode and colors (foreground, background, and border), and the bell.
- Read the current time and date for display by your program. You can also use the current time value to calculate elapsed time or generate a waiting period or delay.
- Transfer data to or from peripheral cards.
- Call some of the Monitor subroutines from full native mode.

Tool Set Interactions

As mentioned earlier in this chapter, some tool sets involve other tool sets in their work. Recall, for example, that the Menu Manager relies on QuickDraw II to draw menus. Similarly, the Window Manager relies on QuickDraw II to draw its border elements and on the Control Manager to manage its border elements. Although the way tool sets interact may appear to be a complex and puzzling web, it will become clearer once you have gained some programming experience.

While the tool sets can't be ranked in order of importance, they can be

ranked in order of *involvement* — according to how other tool sets depend on them. Figure 6-5 is an attempt to show the interdependencies of the tool sets. Here, the sets toward the top depend on the ones below them. You can, if you wish, use this figure as a guide for your programming. As a rule of thumb, for a given tool set to work properly, you should have activated all the tool sets shown below it.

Using Tool Calls in Programs

Tool calls are generally macro calls, so you create a program that makes tool calls just as you create one that calls any of the general-purpose macros described in Chapter 5. That is:

1. Create the source program using the editor. At the beginning of the program, enter an *MCOPY lib-file* directive, where *lib-file* is the name of the macro library file that will contain tool call macros and other macros used in the program.
2. Enter each tool name at the place in the program where you want to use it.

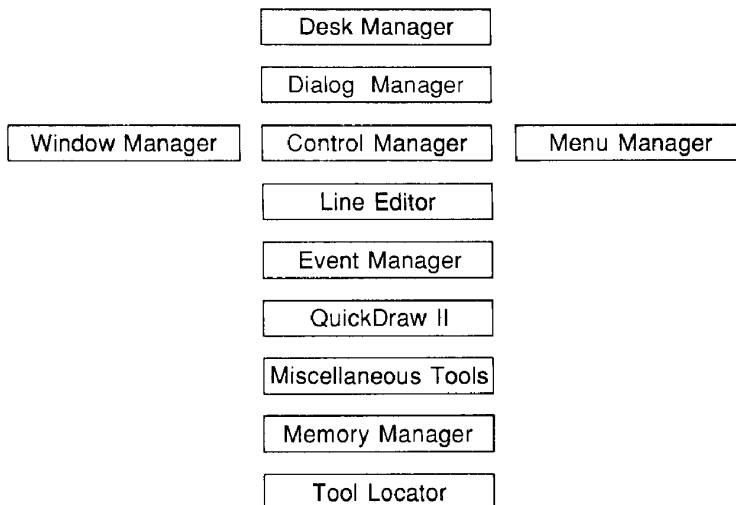


Figure 6-5

3. Save the program on disk.
4. The tool call macros are in the /APW/MACROS directory on the Programmer's Workshop disk. Therefore, to construct the macro library, enter a command of the form:

```
MACGEN source-file lib-file /APW/MACROS/=
```

5. Assemble and link your program with

```
ASML source-file
```

if the program contains a KEEP directive, or else with:

```
ASML source-file KEEP=out-file
```

Making Tool Calls

There are hundreds of tool calls, and they are all described in the two-volume *Apple IIGS Toolbox Reference*. Apple's software designers have given the tools simple, one-word names to make them easy to understand and remember. For example, the calls `WindStartup` and `WindShutDown` activate and deactivate the Window Manager. Other calls to the Window Manager include `NewWindow`, `CloseWindow`, `SetWTitle`, `SetFrameColor`, `ShowWindow`, `HideWindow`, and `MoveWindow`.

That's how the tool names appear in the *Toolbox Reference*. In assembly language programs, you must precede each tool name with an underscore (`__`) character. For example, you must enter `WindStartup` as `__WindStartup`, `WindShutDown` as `__WindShutDown`, and so on.

Calling Conventions

Some tools (`WindShutDown`, for example) require no input values and produce no result value, but these are the exception rather than the rule. Most tools require input values or produce a result, or both. Tools that pass values to and from the application use the *stack* to do so.

This means that before you call a tool that requires input values, you must push those values onto the stack. It also means that before you call a tool that produces a result value, you must reserve space for it on the stack. Finally, if a tool both requires inputs and produces a result, you must do

both operations: reserve space for the result, *then* push the inputs onto the stack. Hence, calling a tool can take up to four steps:

1. Push space on the stack for the result (if any).
2. Push the input parameters onto the stack.
3. Make the tool call.
4. Pull the result (if any) from the stack.

Inputs and the result are usually 2- or 4-byte values, so the standard way to reserve stack space or push data is by using the M16.UTILITY macros *PushWord* and *PushLong*. For example, you would use the following kind of sequence to call a tool that requires three inputs (2 words and a long word, in this case) but does not return a result:

```
PushWord    param1      ; Push a 2-byte value,
PushWord    param2      ; then another
PushLong    param3      ; Push a 4-byte value
_ToolName
```

Conversely, the following sequence calls a tool that returns a 2-byte result, but requires no input values:

```
PushWord    #0          ; Reserve space for a 2-byte
                        ; result
_ToolName
```

Here, *PushWord* pushes a 0, but any value would do. After all, you're simply reserving space; the value you push is never used.

Finally, this sequence calls a tool that takes three inputs and returns a 2-byte result:

```
PushWord    #0
PushWord    param1
PushWord    param2
PushLong    param3
_ToolName
```

Again, because the stack operates in first-in/last-out fashion, the push that reserves space for the result must *always* come first. After the tool routine

has removed the inputs from the stack, it is left with space in which to insert the result.

Words, Integers, and Pointers

Tool calls work with two kinds of 2-byte values; unsigned values are called *words*, while signed values are called *integers*. The most common kind of 4-byte value (or longword) is a memory address or *pointer*. Keep those terms in mind as you read the rest of this book.

General Structure of an Application Program

I have no way of knowing the details of the programs you want to create, but every Apple IIGS program that uses the Toolbox must have the same general elements. Specifically, every program must:

1. MCOPY the macro file for your program.
2. Set the data bank equal to the program bank, so you can use absolute addressing to access any data that may be in your program.
3. Start the Tool Locator, Memory Manager, and Miscellaneous Tools. Miscellaneous Tools and every other tool set makes the carry flag 1 if it cannot be started for some reason. Therefore, you must exit immediately if a Startup call returns $C = 1$.
4. Some tool sets need working space in bank 0. Request it from the Memory Manager.
5. Start QuickDraw II and any other ROM-based tool sets your application uses.
6. Read any needed RAM-based tools from disk.
7. Start the RAM-based tools.
8. Run your application.
9. Shut down all the tool sets you used.

Using the Program Bank as the Data Bank

Setting the data bank equal to the program bank takes only two instructions:


```

phk          ; Push the program bank register onto the
              ; stack,
plb          ; then pull it into the data bank register

```

Starting the Tool Locator and Memory Manager

To start the Tool Locator and Memory Manager, enter:

```

_TLStartup
PushWord #0  ; Reserve space for result (ID number)
_MMStartup

```

The Memory Manager returns a word-size program identification number that you must use to request bank 0 space for the tools that need it. Save this number with:

```

pla          ; Pull program ID from the stack
sta  MyID    ; and store it in memory

```

Start Miscellaneous Tools with a sequence of the form:

```

_MTStartup
ldx #3
jsr PrepareToDie

```

The *PrepareToDie* subroutine should check the carry flag, and exit the program with an error message if carry is 1. Here, the X register receives the error code (3, in this case) that is to be included in the error message.

Allocating Working Space in Bank 0

The tool sets listed in Table 6-1 need working space in bank 0. Note that QuickDraw II needs three pages (where a page is 256 or \$100 bytes), while the rest need only one page. To obtain this space from the Memory Manager, you must issue a NewHandle call.

NewHandle takes four inputs from the stack: a longword that specifies how many bytes you need, a word that contains your program ID (the one you stored in MyID), a word that specifies the type of memory block being requested (fixed or relocatable), and a longword that specifies the desired

Table 6-1

Tool Set	Bytes Required
QuickDraw II	\$300
Event Manager	\$100*
Control Manager	\$100**
Menu Manager	\$100
Line Editor	\$100
Font Manager	\$100
Print Manager	\$100
Sound Manager	\$100
Standard File Operations	\$100
SANE	\$100

*The Event Manager shares its page with the Window Manager.

**The Control Manager shares its page with the Dialog Manager.

location of the block (if any). NewHandle returns a longword *handle* on the stack, so you must reserve stack space for it. (Recall from the “Memory Manager” section that a handle is a pointer to a pointer. In this case, the handle points to the address of the memory block.) For example, to request seven pages in bank 0, enter:

```

PushLong    #0          ; Reserve stack space for result
              ; (handle)
PushLong    $$700       ; Request 7 pages
PushWord    MyID        ; Program ID
PushWord    $$C005      ; Locked, fixed location, fixed
              ; bank
PushLong    #0          ; Location
_NewHandle

```

After that, you must retrieve the handle from the stack and store it in the direct page (the 65816's zero page). QuickDraw and other tool sets also need to know the address of their direct page space, so you must get the pointer to it using indirect addressing. Instructions that do this are:

```

pla          ; Pull handle from stack
sta    0     ; and store it in direct page
pla
sta    2

lda    [0]   ; Get pointer to memory block
sta    4     ; and store it at loc. 4 of DP

```

Note that although the pointer is a 3-byte address, the *sta 4* instruction only stores the lower 2 bytes. It's unnecessary to store the bank number because the direct page is always in bank 0.

Starting ROM-Based Tool Sets

In addition to the Tool Locator, Memory Manager, and Miscellaneous Tools, nine other tool sets — including QuickDraw II and the Event Manager — are built into ROM. (I'll provide the full list shortly.) Just as you start the Tool Locator, Memory Manager, and Miscellaneous Tools using `__TLStartup`, `__MMStartup`, and `__MTStartup`, you start every other tool set using a call that ends with "Startup." For example, `__QDStartup` starts QuickDraw II, `__EMStartup` starts the Event Manager, and so on.

I describe each Startup call as I discuss its tool set throughout the rest of the book. Each Startup call sets the carry flag to 1 if an error occurred. Thus, you should follow the Startup with a *jsr PrepareToDie* sequence, as I did when starting Miscellaneous Tools.

Reading RAM-Based Tools from Disk

The RAM-based tool sets are in the System Disk's /SYSTEM/TOOLS sub-directory, and they are loaded into RAM when you boot up. To use any RAM-based tool set in a program, you must tell the computer which ones you want by calling LoadTools. The disk also contains additional tool calls for some of the ROM-based tool sets; to use these extra tools, you must specify their tool sets to LoadTools, too.

The generalized calling sequence for LoadTools is:

```
PushLong ToolTablePtr
_LoadTools
```

where *ToolTablePtr* points to a table of the form:

```
ToolTable    dc i'NumTools'
              dc i'ToolNum1, MinVersion'
              dc i'ToolNum2, MinVersion'
              .
              .
              dc i'ToolNumN, MinVersion'
TTEnd        anop

TableSize    equ  TTEnd-ToolTable-2
NumTable     equ  TableSize/4
```

Here, *NumTools* specifies the number of entries in the rest of the table. The equates at the end make the assembler calculate its value.

Each entry consists of the tool set's identification number and the number of the earliest or "minimum" version of that set with which your program can work. (More about minimum version numbers shortly.) Table 6-2 lists the identification numbers Apple has assigned to the tool sets and indicates where each resides, in ROM or on disk.

The Apple IIGS is an evolving computer, and Apple will issue updates to the tool sets that add new tool calls or perhaps replace or change existing tool calls. The minimum version (*MinVersion*) parameter in each tool table

Table 6-2

ROM-Based Tools	
1	Tool Locator
2	Memory Manager
3	Miscellaneous Tools
4	QuickDraw II
5	Desk Manager
6	Event Manager
7	Scheduler
8	Sound Manager
9	Apple Desktop Bus
10	SANE
11	Integer Math Tools
12	Text Tools
13	(Used internally)
RAM-Based Tools	
14	Window Manager
15	Menu Manager
16	Control Manager
17	Loader
18	High-Level Printer Driver
19	Low-Level Printer Driver
20	Line Editor
21	Dialog Manager
22	Scrap Manager
23	Standard File Operations
24	Disk Utilities
25	Note Synthesizer
26	Note Sequencer
27	Font Manager

entry lets your program adapt to changes within tool sets. That is, it lets you specify the earliest version of a tool set with which the program can work.

Tool set version numbers are of the form XX.xx, where XX and xx are the major and minor version numbers. Hence, a tool set whose version number is 1.02 has a major version number of 1 and a minor version number of 02. In the tool table, you must enter the minimum version number as a word value, where the upper byte gives the major version number and the lower byte gives the minor version number. For example, to specify version 1.02 as the minimum version number, enter \$0102 in the tool table.

Every tool set includes a "Version" call that returns the current version number (QuickDraw II's call is QDVersion, for example), and you would use them to obtain the number for the tool sets you're now using. However, if you don't care which version the user has, or you only intend to use the program on your own computer, you can enter \$0100 for each minimum version number. For example, the following LoadTools call activates eight RAM-based tool sets:

```
InitStuff START
    using GlobalData
    . .
    . .
    PushLong #ToolTable
    _LoadTools
    . .
    . .
    END

GlobalData DATA
    . .
    . .
    ToolTable dc i'NumTools'      ;No. of tool sets in
                                ; the table
                dc i'4,$0100'      ;QuickDraw
                dc i'5,$0100'      ;Desk Manager
                dc i'6,$0100'      ;Event Manager
                dc i'14,$0100'     ;Window Manager
                dc i'15,$0100'     ;Menu Manager
                dc i'16,$0100'     ;Control Manager
                dc i'20,$0100'     ;Line Editor
                dc i'21,$0100'     ;Dialog Manager
```

```

TTend      anop

TableSize  equ  TTend-ToolTable-2
NumTools   equ  TableSize/4
.
.
.
END

```

Note that the LoadTools call is in a code segment (InitStuff), whereas the ToolTable is in a data segment (GlobalData). This is the proper way to separate instructions and data in a IIGS application program. Note also that InitStuff must declare that it is *using GlobalData*, so the assembler knows where to look for the ToolTable.

Of course, for LoadTools to work correctly, the tool sets it selects must be on disk. If they aren't, LoadTools sets the carry flag to 1 and loads a ProDOS error code into the accumulator. An error code of \$45 indicates "Volume directory not found," which means, in this case, that ProDOS couldn't find the tools in memory; any other code indicates some other ProDOS error.

If LoadTools returns carry equal to 1, there are several things your program can do. It can, of course, simply give up — that is, display a message and break using PrepareToDie. However, that's a rather harsh way to punish a user who simply booted with the wrong disk. It's more reasonable to tell the user to insert the disk, then try loading the tools again. If LoadTools still returns a carry of 1, the program should shut down the tools and exit.

Example 6-1 shows instructions that do all this (and assumes you want the eight tool sets I loaded in the preceding example). In fact, it displays not just a prompt, but a pseudo *dialog box*. The box contains the prompt and two buttons that let the user signal that he or she has either inserted the disk (OK) or wants to quit the program (Cancel).

This program fragment has three code segments (Progame, InitStuff, and MountBookDisk) and one data segment (GlobalData). The main segment, Progame, contains a call to an InitStuff subroutine and a branch instruction that exits if InitStuff cannot load the tools.

InitStuff calls LoadTools, which checks whether the tools listed in the ToolTable are in memory; if not, it attempts to load them from disk. If the tool sets can't be loaded (carry is 1), InitStuff checks the accumulator for the "Volume directory not found" code, \$45. This value makes the program branch to DoMount; any other makes it exit with PrepareToDie.

At DoMount, the program calls a MountBookDisk subroutine. To

Example 6-1

```

Progame START
    ..
    ..
    jsr  InitStuff      ;Initialize the tools
    bcs  AllDone        ;Quit if tools couldn't be initialized
    ..                ;Otherwise, continue here
    ..
    END
InitStuff START
    using GlobalData
    ..
    ..
LoadAgain PushLong  #ToolTable
    _LoadTools
    bcc  ToolsLoaded    ;Error?

    cmp  #VolNotFound   ; Yes.
    beq  DoMount        ;If it's $45, get user to insert disk
    sec                      ;Otherwise, exit
    ldx  #$FE
    jsr  PrepareToDie
DoMount  anop
    jsr  MountBootDisk  ;Get user to mount the disk
    cmp  #1             ;Has correct disk been mounted?
    beq  LoadAgain     ; Yes. Try loading again
    sec                      ; No. Set error flag
    rts                ; and return to caller
ToolsLoaded  anop      ;Tools have been loaded
    ..
    ..
    END
MountBootDisk START

    _Set_Prefix SetPrefixParams
    _Get_Prefix GetPrefixParams

    PushWord #0          ;Space for result
    PushWord #195        ;Column position for dialog box
    PushWord #30         ;Row position for dialog box
    PushLong #PromptStr  ;Prompt at top of dialog box
    PushLong #VolStr     ;Volume name string
    PushLong #OKStr      ;String in Button 1
    PushLong #CancelStr  ;String in Button 2
    _TLMountVolume
    pla                  ;Obtain the button number
    rts                  ; and return to caller
PromptStr str 'Please insert the disk.'
VolStr    ds 16
OKStr     str 'OK'
CancelStr str 'Shutdown'

GetPrefixParams dc i'7'
                dc i4'VolStr'

```

176 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

```
SetPrefixParams dc i'7'
                dc i4'BootStr'
BootStr    str '*/'
            END

GlobalData DATA

ToolTable dc i'NumTools' ;No. of tool sets in the table
          dc i'4,$0100' ;QuickDraw
          dc i'5,$0100' ;Desk Manager
          dc i'6,$0100' ;Event Manager
          dc i'14,$0100' ;Window Manager
          dc i'15,$0100' ;Menu Manager
          dc i'16,$0100' ;Control Manager
          dc i'20,$0100' ;Line Editor
          dc i'21,$0100' ;Dialog Manager
TTEnd      anop

TableSize equ TTEnd-ToolTable-2
NumTools equ TableSize/4

VolNotFound equ $45 ;ProDOS error
            END
```

begin, MountBookDisk calls SetPrefix and GetPrefix to read the current volume name into a 16-byte location called VolStr (Volume String). Then it calls TLMountVolume, a tool that displays a dialog box and waits for the user to “press” one of two buttons using the mouse. By pressing *OK* (button 1), the user indicates the System Disk has been inserted; by pressing *Shut-down* (button 2), he or she terminates the program.

Starting RAM-Based Tool Sets

Like ROM-based tool sets such as QuickDraw II, the RAM-based tool sets have calls that end in “Startup.” I will describe each Startup call as I discuss its tool set throughout the rest of the book.

Shutting Down the Tool Sets

Just as you must start up tool sets before you can use them, you must shut them down when you finish using them. Each tool set has a call ending with “ShutDown.” For example, __QDShutDown shuts down QuickDraw II, __MenuShutDown shuts down the Menu Manager, and so on. None of the ShutDown calls return a result value, and only the Memory Manager call, __MMShutDown, requires an input. To shut down the Memory Manager, you must push your program’s I.D. number (the number produced by __MMStartup) onto the stack. To do this, enter calls of the form:

```
PushWord MyID
__MMShutDown
```


I'll give the entire shut-down sequence later.

Leaving the Program

To leave the program and return to the Program Launcher or the *Programmer's Workshop*, you must give a ProDOS Quit command. The tool call that does this, Quit, has the general form:

```
_Quit QuitParams
```

where *QuitParams* is defined by:

```
QuitParams  dc  i4'0'
             dc  i'$4000'
```

Here, the four-byte 0 value tells ProDOS to return to the calling program (APW shell or Program Launcher), while the word-size \$4000 value tells it to keep the program in memory. Having a program in memory allows you to run it later without reloading it from disk.

Quit can produce four different errors (“Path not found”, “File not found”, “Out of memory”, or “Not an executable system file”) that leave the computer in an unknown state. For this reason, you should follow the Quit call with a *brk \$F0* instruction to force an exit.

Generalized Program Model

In the preceding section, I listed and described the elements that every program must have. I will now pull all of these elements together in a generalized program *model* that includes the “boilerplate” most programs need. Once you have created the model file, you can use it as a starting point to produce every program you write.

Example 6-2 shows the model. Although it's rather long, realize that (fortunately) you must only enter it once.

The model contains many tool calls that I have not yet described. Please bear with me — if I described everything now, you would probably become both utterly confused and totally bored. Look over the listing, but don't try to understand every single detail. Rest assured that I will describe these calls eventually, at places where I use them throughout the rest of the book.


```

ToolTable      dc i'NumTools'          ;No. of tool sets in table
               dc i'4,$0100'           ;QuickDraw
               dc i'5,$0100'           ;Desk Manager
               dc i'6,$0100'           ;Event Manager
               dc i'14,$0100'          ;Window Manager
               dc i'15,$0100'          ;Menu Manager
               dc i'16,$0100'          ;Control Manager
               dc i'20,$0100'          ;LineEdit
               dc i'21,$0100'          ;Dialog Manager
TTEnd          anop

TableSize      equ TTEnd-ToolTable-2
NumTools       equ TableSize/4

VolNotFound    equ $45                ;ProDOS error

END

```

```

;*****
;
; InitStuff
;
; Initializes tool sets, gets space in bank 0 for use as zero
; page by tool sets that need it, and ensures that RAM-based
; tools are in memory.
;
;*****

```

```

InitStuff      START
               using GlobalData

```

```

; Initialize the Tool Locator, Memory Manager, and Miscellaneous
; Tools.

```

```

      _TLStartup          ;Tool locator

      PushWord #0          ;Memory Manager
      _MMStartup

      pla                 ;Memory Manager returns program's ID
      sta MyID

      _MTStartup          ;Misc. Tools
      ldx #3
      jsr PrepareToDie

```

```

; Get some space for the direct page we need. QuickDraw needs
; three pages and the Event Manager, Control Manager, Line
; Editor, and Menu Manager each need one page.

```

```

      PushLong #0          ;Space for handle
      PushLong #$700       ;Seven pages
      PushWord MyID        ;Owner
      PushWord #$C005      ;Locked, fixed, fixed bank
      PushLong #0          ;Location
      _NewHandle
      ldx #$FF
      jsr PrepareToDie

      pla                 ;Read handle and store in dp
      sta 0

```

180 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

```

        pla
        sta 2

        lda [0]                ;Get dp location from handle
        sta 4                  ; and store at loc 4 of dp

; Initialize QuickDraw

        pha                    ;Use dp obtained from handle
        PushWord #ScreenMode   ;Mode = 640
        PushWord #160          ;Max size of scan line (in bytes)
        PushWord MyID
        _QDStartup
        ldx #4
        jsr PrepareToDie

; Initialize Event Manager

        lda 4                  ;dp to use = QD dp + $300
        clc
        adc #$300
        sta 4
        pha
        PushWord #20           ;Queue size
        PushWord #0            ;X clamp left
        PushWord #MaxX         ;X clamp right
        PushWord #0            ;Y clamp top
        PushWord #200          ;Y clamp bottom
        PushWord MyID
        _EMStartup
        ldx #6
        jsr PrepareToDie

;-----
;
; Load the RAM-based tools I need.

LoadAgain    PushLong #ToolTable
              _LoadTools
              bcc ToolsLoaded

              cmp #VolNotFound
              beq DoMount
              sec
              ldx #$FE
              jsr PrepareToDie

DoMount      anop
              jsr MountBootDisk
              cmp #1
              beq LoadAgain

              sec
              rts

; The tools have been loaded. Initialize the Window Manager, Control
; Manager, LineEdit, Dialog Manager, Menu Manager, and Desk Manager.

ToolsLoaded  anop
              PushWord MyID          ;Window Manager
              _WindStartup

```

```

ldx #14
jsr PrepareToDie

PushLong #0                ;Prepare screen for windows
 RefreshDesktop

PushWord MyID              ;Control Manager
lda 4                      ;DP to use = EM DP + $100
clc
adc #$100
sta 4
pha
_CtlStartup
ldx #16
jsr PrepareToDie

PushWord MyID              ;LineEdit
lda 4
clc
adc #$100
sta 4
pha
_LEStartup
ldx #20
jsr PrepareToDie

PushWord MyID              ;Dialog Manager
 DialogStartup
ldx #21
jsr PrepareToDie

PushWord MyID              ;Menu Manager
lda 4
clc
adc #$100
pha
_MenuStartup
ldx #15
jsr PrepareToDie

_DeskStartup              ;Desk Manager

clc                        ;Clear the carry flag
rts                        ; and return

END

;*****
;
; Event Loop
;
; This is the subroutine that interacts with the user.
;
;*****

EventLoop      START
               using GlobalData
               rts
               END

```

182 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

```
*****
*
* Prepare To Die
*
* Dies if carry is set.
*
* Error number to display is in X register.
* Assumes that Miscellaneous Tools is active.
*
*****
```

```
PrepareToDie  START
               bcs RealDeath      ;Carry = 1?
               rts                 ; No. Return to caller

RealDeath     phx                  ; Yes. Goodbye, program.
               PushLong #DeathMsg
               _SysFailMgr

DeathMsg      str 'Could not handle error '

               END
```

```
*****
;
; Mount Boot Disk
;
; Tells the user to insert the disk that contains the RAM-based
; tools.
;
*****
```

```
MountBootDisk  START

               _Set_Prefix SetPrefixParams
               _Get_Prefix GetPrefixParams

               PushWord #0          ;Space for result
               PushWord #195       ;Column position for dialog box
               PushWord #30        ;Row position for dialog box
               PushLong #PromptStr  ;Prompt at top of dialog box
               PushLong #VolStr     ;Volume name string
               PushLong #OKStr      ;String in Button 1
               PushLong #CancelStr  ;String in Button 2
               _TLMountVolume

               pla                  ;Obtain the button number
               rts                 ; and return to caller

PromptStr      str 'Please insert the disk.'
VolStr         ds 16
OKStr          str 'OK'
CancelStr      str 'Shutdown'

GetPrefixParams dc i'7'
               dc i4'VolStr'
SetPrefixParams dc i'7'
               dc i4'BootStr'

BootStr        str '*/'

               END
```

What's In the Model

The model has one data segment (*GlobalData*) that contains the program's variables. There are also five code segments.

The *Model* segment contains the program's starting and ending points, as well as global equates that are used throughout the program. It contains a call to *InitStuff*, which starts the tool sets the program uses. From *MTStartup* on, each Startup sequence ends by calling *PrepareToDie* with the tool set number in the X register. If carry is 1 (the startup failed), *PrepareToDie* exits with an error message.

InitStuff also calls *LoadTools* to read the RAM-based tools from disk. If *LoadTools* is unsuccessful, a *MountBootDisk* subroutine prompts the user to insert the correct disk. If *InitStuff* still cannot load the tools, it sets the carry flag to 1 and returns to the Model segment, where a Branch on Carry Set (bcs) instruction closes down the program.

If all tools have been successfully initialized, the program calls an *EventLoop* subroutine (empty at the moment) to interact with the user. When the user quits, Model shuts down the tool sets and makes a ProDOS Quit call to return to the system program.

Creating the Model File

Create the model file now, using the editor, and name it MODEL.SRC (i.e., enter **edit model.src**). Before starting the editor, be sure to give an **asm65816** command so that MODEL.SRC will be an assembler source code file.

When you finish, print a copy of MODEL.SRC and compare it closely with Example 6-2. If you find any typographical errors or omissions, correct them using the editor.

Validating MODEL.SRC

To help ensure that you have entered the MODEL.SRC file correctly, you should assemble and link it. Before you can assemble, however, you must create the MODEL.MACROS file using MACGEN. To do this, enter the command:

```
macgen model.src model.macros /apw/libraries/  
ainclude/ml6.=
```

If you mistyped one or more tool calls, MACGEN displays their names and asks for the name of a file to search. That's your cue to press Esc and leave MACGEN so you can make the corrections.

If MACGEN runs without errors, assemble and link the model file by entering:

```
asml model.src keep=model`
```

The *keep* option is necessary here because the model does not contain a KEEP directive. (You need a KEEP in *one* of these places to generate the object and load files.)

Putting KEEP on the command line is handy because it lets you create multiple versions of a program. For example, suppose you have developed a program called MYPROG and it doesn't do quite what you want. You can load MYPROG.SRC into the editor, make the required changes, then save it as, say, MYPROG1.SRC. Now you have two different versions of the same program! When you finally get a version that works, delete all the others and rename the working version with the original name (MYPROG, in this case).

Providing that the assembler and linker report no errors, the preceding ASML command produces three new files: object (OBJ) files MODEL.ROOT and MODEL.A, and the shell load (EXE) file MODEL. These files are simply by-products of the verification procedure. They aren't needed, so you can delete them. However, retain MACGEN's output, MODEL.MACROS, because it's a handy macro library file for MACGENs you perform on behalf of your application programs.

Using the Model

To create an application program using MODEL.SRC, do the following:

1. With the *Workshop* prompt on the screen, enter **edit model.src** to start the editor with the model file.
2. Do a Search and Replace Down (OpenApple-J) command to replace *Model* with the name of your program.
3. Delete tool calls and instructions your program doesn't need. The editor's Cut (OpenApple-X) command is handy for deleting groups of lines.

If there is an entire tool set that you don't need, follow these rules:

- Delete the tool set's Startup sequence from the InitStuff segment and its ShutDown call from the Model segment.
 - If the tool set is RAM-based, delete its entry from the ToolTable list in the GlobalData segment.
 - If the tool set is one that requires space in bank 0, change the operand for the second PushLong in the calling sequence for __NewHandle (in the InitStuff segment).
4. If your program requires a tool set that is not in the model, do the following:
 - Insert its Startup and ShutDown sequences in the InitStuff and Model segments, respectively.
 - If the tool set is RAM-based, insert its entry in the ToolTable list in the GlobalData segment. For the tool set number, see Table 6-2 under "Reading RAM-Based Tools from Disk."
 - The Sound Manager, Standard File Operations, and SANE each require a page in bank 0. To use one of these tool sets, you must add \$100 to the operand for the second PushLong in the __NewHandle calling sequence (this is in the InitStuff segment).
 5. Insert the data, instructions, and tool calls for your program.
 6. Leave the editor and select the N (Save to a new name) option to save the file with the name you want. Then select the E (Exit without updating) option to make the APW prompt reappear.
 7. To generate the macro library file for your program, enter a command of the form:

```
macgen myprog.src myprog.macros model.macros
/apw/libraries/ainclude/ml6. =
```

where MODEL.MACROS is the macro library file produced by the MACGEN you performed to verify MODEL.SRC. (If you deleted this library, omit its name from your MACGEN command. MACGEN will then create MYPROG.MACROS from scratch.)

Copying Between Programs

You needn't start every new program using the model, of course; you may want to construct a new program based on a similar program that already exists. The editor's copy and paste (OA-C and OA-V) commands are handy for copying code sequences between programs.

As you may recall, the editor always stores copied text *on disk*, which means that it remains intact when you switch programs — or turn the computer off, for that matter. Thus, to reproduce an existing instruction sequence in a new program, simply highlight it (with OA-C and the arrow keys), save it (Return), exit (Ctrl-Q), load the new program (L), and paste the copy where you want it (OA-V).

Tool Locator, Memory Manager, and Miscellaneous Tools Calls

The model program in this chapter contains many tool calls that I have not yet covered. As promised, I will discuss them later. However, I *have* described some calls to the Tool Locator, Memory Manager, and Miscellaneous Tools; Table 6-3 summarizes them.

Start-Up and Shut-Down Sequences

The Apple IIGS requires its major tool sets to be started in the following order:

- Tool Locator
- Memory Manager
- Miscellaneous Tools
- QuickDraw
- Event Manager
- Window Manager
- Control Manager
- Line Editor
- Dialog Manager
- Menu Manager
- Desk Manager

Font Manager, Standard File, and other tool sets not listed here.

Table 6-3

Tool Locator	
__TLStartup	Start the Tool Locator
Call with: __TLStartup	
Result: None	
__TLShutDown	Shut down the Tool Locator
Call with: __TLShutDown	
Result: None	
__LoadTools	Load RAM-based tools from boot disk
Call with: PushLong ToolTablePtr	;Pointer to tool table
__LoadTools	
Result: None	
Note: ToolTablePtr points to	
dc i'NumTools'	
dc i'ToolNum1,MinVersion'	
dc i'ToolNum2,MinVersion'	
..	
..	
dc i'ToolNumN, MinVersion'	
__LoadOneTool	Load specified tool set from boot disk
Call with: PushWord #ToolNum	;Tool set number
PushWord #MinVersion	;Minimum version number
__LoadOneTool	
__TLMountVolume	Display a dialog box with a prompt
Call with: PushWord #0	;Space for result
PushWord WhereX	;Column pos. for dialog box
PushWord WhereY	;Row pos. for dialog box
PushLong PromptStr	;Prompt at top of dialog box
PushLong VolStr	;Volume name string
PushLong But1Str	;String in Button 1
PushLong But2Str	;String in Button 2
__TLMountVolume	
Result: Button number (word)	

Memory Manager

__MMStartup	Start the Memory Manager
Call with: PushWord #0	;Space for result
__MMStartup	
Result: Program ID (word)	
__MMShutDown	Shut down the Memory Manager
Call with: PushWord ProgID	;Program ID
__MMShutDown	
Result: None	

Table 6-3 (cont.)

Memory Manager (cont.)		
<hr/>		
__NewHandle		Allocate a block in memory
Call with:	PushLong # ;Space for result	
	PushLong <i>BlockSize</i> ;Block size in bytes	
	PushWord <i>Owner</i> ;Program ID	
	PushWord <i>Attributes</i> ;Attributes of block	
	PushLong <i>Location</i> ;Location	
	__NewHandle	
Result:	Handle (longword)	
__DisposeHandle		Dispose of a block and its handle
Call with:	PushLong <i>Handle</i> ;Handle	
	__DisposeHandle	
Result:	None	
__DisposeAll		Discard all blocks and handles
Call with:	PushWord <i>Owner</i> ;Program ID	
	__DisposeAll	
Result:	None	
<hr/>		
Miscellaneous Tools		
__MTStartup		Start the Miscellaneous Tools
Call with:	__MTStartup	
Result:	None	
__MTShutDown		Shut down the Miscellaneous Tools
Call with:	__MTShutDown	
Result:	None	
__SysFailMgr		Exit and display an error message
Call with:	PushWord <i>DeathNum</i> ;Error code, follows message	
	PushLong <i>MessagePtr</i> ;Pointer to message string	
	__SysFailMgr	
Result:	None	
<hr/>		

Remember that before starting the Window Manager, you must do a LoadTools call to load the RAM-based tools your program uses. Remember also that you must allocate memory (with a NewHandle call) for the tool sets that need it.

Shut down the tool sets in the following order:

Desk Manager

Font Manager, Standard File, and other tool sets not listed here.

Menu Manager

Window Manager

Control Manager
 Dialog Manager
 Event Manager
 Line Editor
 QuickDraw II
 Miscellaneous Tools
 Memory Manager (after freeing all allocated memory)
 Tool Locator

Common Programming Errors

No matter how carefully you program, you may eventually run into a situation where the computer beeps and the program “crashes.” The beep indicates that the IIGS has executed a BRK instruction, which sends the processor into the Monitor. When that happens, all you can do is reboot. Needless to say, having a program crash gives you a helpless feeling — take it from one who knows.

What should you do when a program crashes? To begin, you should look for obvious errors in the program listing. Don’t bother looking for typographical errors; the assembler will have reported them when you assembled the program. Instead, look for errors related to tool calls. Among the most common are:

1. Calling a tool without activating it, *starting* its tool set, or misplacing the set’s StartUp call in the initialization sequence (see the preceding section).
2. Entering too few or too many input parameters, or entering them in the wrong order.
3. Pushing a word onto the stack when the tool expects a longword, or vice versa.
4. Pushing data onto the stack when the tool expects an address. Remember, *PushLong Loc* pushes the data stored at Loc, while *PushLong #Loc* pushes Loc’s address (i.e., it pushes a pointer to Loc).
5. Omitting a *PushWord #0* or *PushLong #0* call that reserves stack space for the result.

6. Forgetting to pull a result off the stack, or pulling 4 bytes when the tool returned only 2 (or vice versa).
7. Trying to exit the program without shutting down an active tool set. This error is easy to identify. The program will run fine, but crash upon exiting.
8. Failing to allocate enough working space in bank 0 with your New-Handle call. (See Table 6-1 for the memory requirements of the various tools sets.)

If the error isn't obvious from the program listing (or you don't want to take time to search the listing), trace through the program using the debugger, to see where the break occurred. The debugger displays instructions until the program makes a QDStartup call, at which point the program takes over and the screen displays your application. To make the instruction trace reappear, type `t` to give the debugger a "change to text mode" subcommand.

CHAPTER 7

Drawing with QuickDraw II

QuickDraw II is the tool set you use to display graphic images on the screen. Other tool sets also use QuickDraw II (or *QuickDraw*, for short) to draw, but they make calls to it behind the scenes, without your realizing it. This chapter introduces the Apple IIGS graphics facilities and guides you through the more important features of QuickDraw.

Graphics Modes

Earlier models of the Apple II provide four screen modes for displaying graphics. In each mode, the screen is divided into a grid of rows vertically and columns horizontally. The more rows and columns the grid has — that is, the higher its *resolution* — the sharper images will appear to the eye. The modes are:

- The *Low-Resolution* mode provides 48 rows and 40 columns, and lets you assign any of 16 colors to each coordinate (intersection of a row and column).
- The *High-Resolution* mode provides 192 rows and 280 columns, and

provides a “palette” of 8 colors. (Six colors, actually, because the palette includes 2 shades of both black and white.)

- The *double low-resolution* mode has (as you may have guessed) 92 rows and 80 columns, with 16 colors.
- The *double high-resolution* mode has 192 rows and 560 columns, also with 16 colors.

The Apple IIGS also supports these modes, although, like other Apple IIs, it has no ROM firmware for either double mode.

Unless you are creating a program to be run on other Apple IIs, you probably won’t use any of these modes, because the Apple IIGS provides something better. Specifically, the IIGS provides two *super high-resolution* graphics modes. One mode divides the screen into 200 rows and 320 columns, and can display any of 16 colors at a given location. The other mode divides the screen into 200 rows and 640 columns, but lets you choose from only 4 colors for a given location.

I’ll get back to these new modes later in this chapter, but first I must discuss the drawing environment in which they operate.

Drawing Environment

On earlier Apple IIs, your drawing space is limited to the coordinates in the screen grid. For example, the largest picture you can draw in regular high-resolution mode is one that occupies no more than 53,760 (280 by 192) screen locations. The Apple IIGS’s super high-resolution graphics modes (or *super hi-res*, for short) can, of course, also display graphics that fill the screen, but with QuickDraw, you can *create* pictures that are much larger than the screen. To do this, you define the pictures in a so-called conceptual drawing space.

Conceptual Drawing Space

You use QuickDraw to create graphics images in a grid that is 32K (that is, 32,768) rows high and 32K columns wide. If you have worked with any other Apple II graphics mode, you might assume that QuickDraw’s grid coordinates are numbered in the same way: with the row 0 and column 0 — or coordinate (0,0) — located at the top left-hand corner of the grid. That isn’t the case here.

QuickDraw’s 32K-by-32K grid consists of four 16K-by-16K quadrants,

with coordinate (0,0) at the center. Figure 7-1 shows how this grid, or *conceptual drawing space*, is numbered. Note that the row and column numbers are positive only in the lower right-hand quadrant; one or both numbers are negative in the other quadrants.

QuickDraw always assumes that you want to start working in the lower right-hand quadrant, so it “maps” onto the screen the image that extends down from and to the right of coordinate (0,0). (Of course, at the outset Quickdraw has no image to display. Your program must create it.) Thus, in the super hi-res 640 mode, the screen displays the graphics image that’s inside an imaginary rectangle whose top left-hand and bottom right-hand corners are at (0,0) and (199,639), respectively. On an Apple RGB color

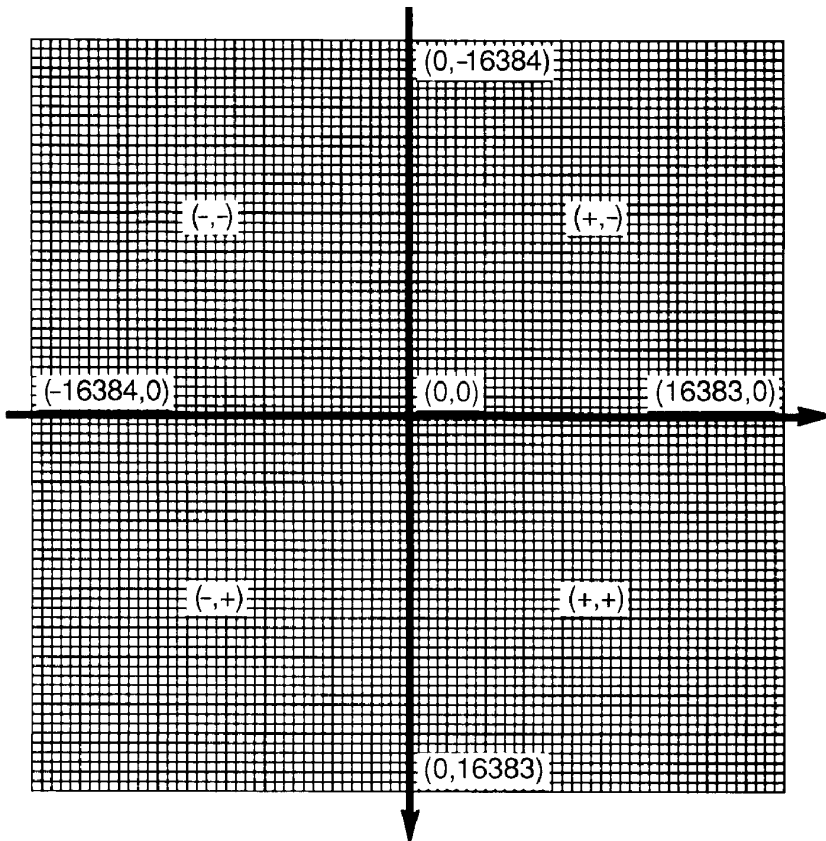


Figure 7-1

monitor, this image area — the entire screen minus the border — is about 6 inches high and 7 3/4 inches wide, so that's the “real” size of the imaginary rectangle.

By comparing these coordinates and dimensions to Figure 7-1, you can begin to realize how large the conceptual drawing space actually is, and what potential it offers for creating pictures. In 640 mode, for example, the screen can display only about five-hundredths of 1 percent of a quadrant, leaving space for almost 2,100 more screenfuls in that quadrant alone! In screen-size terms, this means that theoretically you could create a picture 25 feet high and 25 feet wide!

Be aware that the conceptual drawing space is neither in memory nor on disk. It is simply a coordinate system that is available for into which QuickDraw graphic data may be mapped. If your picture is too large to fit entirely in memory, your program should start by displaying the portion that fits on the screen, and read other portions from disk when the user scrolls to an area that's not visible. More about this later.

Pixels and Points

QuickDraw's drawing space is comprised of picture elements, or *pixels*, that appear as tiny dots on the screen. The screen displays 64,000 pixels in super hi-res 320 mode and 128,000 pixels in 640 mode.

The numeric coordinates of a specific pixel do not refer to the pixel itself, but to the *point* above and to the left of it. A point is simply the place where a row and column intersect — the “address” of a pixel within the conceptual drawing space. To appreciate the difference between a pixel and a point, see Figure 7-2.

Before you can draw something using a QuickDraw tool, you must tell QuickDraw at which point to put it. For example, to draw a rectangle, you must specify the row and column coordinates of its top left-hand and bottom right-hand corners. Here, the top corner indicates the rectangle's starting location and the bottom corner defines its size (see Figure 7-3).

QuickDraw Draws with a Pen

To actually see the rectangle, you must draw it with QuickDraw's *pen*. The pen is involved in every QuickDraw tool call that draws predefined shapes, such as rectangles, arcs, ovals, and lines. It's also involved in displaying text! QuickDraw's pen is more flexible than a regular office pen. Using it, you can draw lines of various thicknesses and even draw patterns.

If you have made the pen one pixel high and one pixel wide — that is,

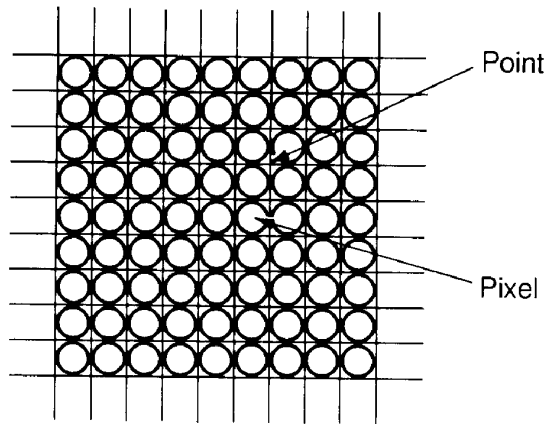


Figure 7-2

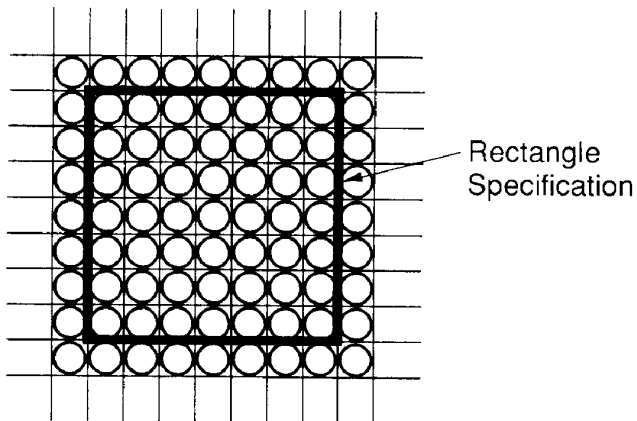


Figure 7-3

a single pixel — QuickDraw will draw the specified rectangle's one-pixel frame *inside* the imaginary rectangle defined by the corner coordinates. Figure 7-4 shows how such a rectangle would appear on the screen. (It would be much smaller, of course. To the eye, a pixel is only about the size of the period at the end of this sentence.)

Before you can use the pen, you must know how to start QuickDraw and shut it down. After all, you can't draw *anything* unless QuickDraw is active.

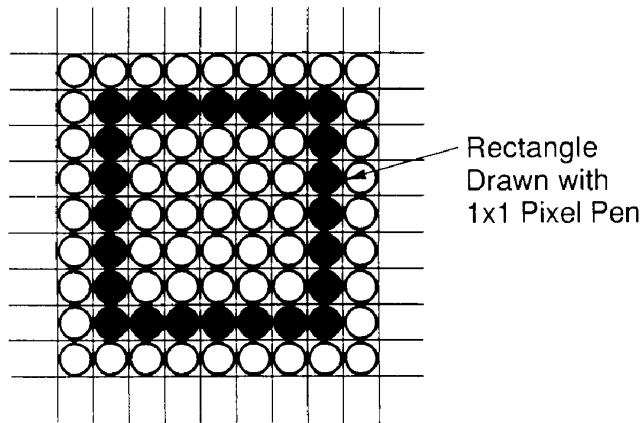


Figure 7-4

Starting and Stopping QuickDraw

To start QuickDraw, you must specify four things: the location of the direct page the Memory Manager has reserved for it, the super hi-res mode in which it is to operate (320 or 640), the maximum number of screen columns it can use for drawing, and the identification number of your program. The generalized sequence of the start-up call is:

```

PushWord DirectPageLoc    ;Direct page location
PushWord MasterSCB        ;Master scan line control byte
PushWord MaxWidth         ;Max. screen width (bytes)
PushWord ProgramID        ;The program's ID number
_QDStartup

```

If you use the program model from Chapter 6 (Example 6-2), you can obtain the direct page location for QuickDraw from the accumulator. Hence, the first call in the QDStartup calling sequence would be *pha*.

With the *master scan line control byte*, you tell QuickDraw which color table (more about color tables later) and which graphics mode to use (320 or 640) for drawing. Figure 7-5 shows the layout of the master scan line control byte. (The “fill” and “interrupt” bits relate to the computer’s video hardware; you can normally set them to zero.)

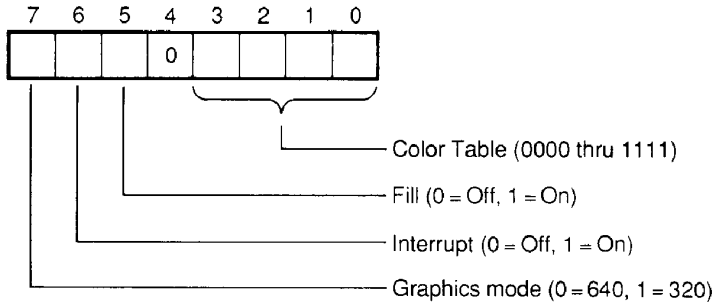


Figure 7-5

Generally, you want to use the default color table (0), so your second start-up call would be either *PushWord #0* (for 320 mode) or *PushWord #\$80* (for 640 mode). Better yet, by defining the scan line control byte with a global equate such as:

```
ScreenMode gequ $80 ;640 mode, no fill
```

as the model does, you can use *PushWord #ScreenMode*.

For most applications, you will want QuickDraw to be able to draw across the entire screen. To make it do this, use *PushWord #0* as your third call.

Finally, the program I.D. is the one returned by the Memory Manager when you start it. The model stores it in a variable named *MyID*, so your program's fourth start-up call would be *PushWord MyID*.

In summary, if you use the model, your QuickDraw start-up sequence is:

```
pha                ;Use DP from handle
PushWord #ScreenMode ;Graphics mode
PushWord #0         ;Use the entire screen
PushWord MyID       ;The program's ID number
__QDStartup
```

As with all tool sets, you must shut down QuickDraw when you finish using it. The call that does this is *__QDShutDown*.

The Pen

The pen with which QuickDraw draws has five properties, or *attributes*: location, size, mode, pattern, and mask. Collectively, they define the *pen state* (called *PenState* in tool calls).

Pen Location

The pen location (or *PenLoc*) specifies the point in the drawing space at which the pen is to sit ready to draw something. Understand, this is simply a location; the pen does not become visible until you actually start drawing with it.

Pen Size

The pen size (*PenSize*) defines the height and width of the pen's imaginary tip. That is, it determines how thick your lines will be. Unlike a regular pen, the tip on QuickDraw's pen is rectangular — so many pixels high by so many pixels wide. If you make the tip 4 pixels high and 2 pixels wide, for example, QuickDraw will draw horizontal lines twice as thick as vertical lines. You can also make the tip square, by specifying the same dimension for height and width. The most common size is 1 by 1 (to draw 1 pixel at a time), but you may want to make it, say, 4 by 4.

Pen size is measured relative to *PenLoc*. Height is measured down from it and width is measured to the right of it.

Pen Mode

QuickDraw provides eight different pen *modes*. The pen mode (*PenMode*) determines how a pen's pixels and its pixel pattern (described next) affect the existing pixels in an image when the pen draws over them.

For most applications, you can use the default mode, *Copy*, in which every pixel the pen draws overwrites (replaces) the pixel that's already at that location in the image. Apple has assigned a word-size number to each pen mode, and *Copy* is represented by \$0000. To learn about the other pen modes, refer to the *Apple IIGS Toolbox Reference*.

Pen Pattern

A regular pen, the kind people use in a home or office, always draws a solid line. You can use QuickDraw's pen in the same way, to draw solid lines or fill in shapes with a solid color. However, sometimes you may want to be

more creative. For example, you may want to draw a curly frame around a picture or give a textured look to the inside of a rectangle or circle.

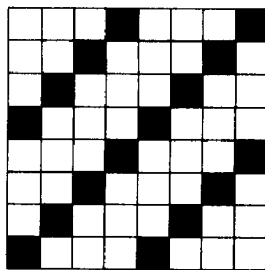
These kinds of artistic effects are easy to produce with QuickDraw; you simply specify the pattern you want the pen to reproduce. After that, it will apply that pattern to every line it draws and every shape it fills in. In short, assigning a pattern makes the pen repeat a specific design as it works its way across the screen, as if it were hanging wallpaper.

The format of the pen pattern (PenPat, in tool calls) differs between the two super hi-res modes. Let's consider the 320 mode first.

In 320 mode, the pen pattern is a data table that specifies the colors for an 8-by-8 square block of pixels. Each of the 64 entries in the table is a 4-bit number, or *nibble*, because in 320 mode, you can display 16 different colors. (Remember, a 4-bit number can have binary values from 0000 to 1111: 16 combinations in all.) Each number in the table indicates the color that the corresponding pixel is to have when the pen passes over it.

Assume for now we're drawing on a monochrome screen, with black images on a white background. In QuickDraw, black has the nibble value \$0 and white's value is \$F. Hence, in defining the table for a pen pattern, you would enter a 0 for pixels that the pen should turn "on" (display as black) and an F for pixels it should leave "off" (retain the white background).

Figure 7-6 shows a simple diagonal pattern you could create, and what it looks like when QuickDraw uses it to draw a rectangular frame. The PenPat table that defines this pattern is:



8x8 Diagonal
Pattern



Diagonal Pattern
As Pen Pattern

Figure 7-6

```

DiagPat  dc h'FFF0FFF0'    ;Row 0 (top)
          dc h'FF0FFF0F'    ;Row 1
          dc h'FOFFF0FF'    ;Row 2
          dc h'OFFF0FFF'    ;Row 3
          dc h'FFF0FFF0'    ;Row 4
          dc h'FF0FFF0F'    ;Row 5
          dc h'FOFFF0FF'    ;Row 6
          dc h'OFFF0FFF'    ;Row 7 (bottom)

```

For this particular pattern, you could use a small pen size and still obtain the diagonal effect. Even a 2-by-2 pen would do the job, although the diagonal might be rather difficult to see. With a 2-by-2 pen, Quickdraw will use the pattern's top two rows to draw horizontal lines and use its two left-hand columns to draw vertical lines.

You can't use a small pen size with all patterns, however. To fill a shape such as a rectangle or circle with a pattern, the pen must be thick enough to cover every significant pixel. Figure 7-7 shows a pattern that requires a thick pen (8-by-8, in this case), and what it looks like when used to fill a block on the screen. The PenPat table that produces this attractive cane-style motif is:

```

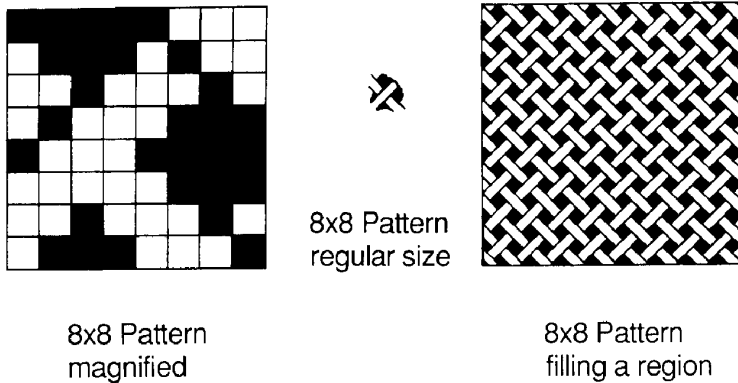
CanePat  dc h'FFFFFF00'    ;Row 0 (top)
          dc h'OFFF0F00'    ;Row 1
          dc h'00F000F0'    ;Row 2
          dc h'0F000FFF'    ;Row 3
          dc h'F000FFFF'    ;Row 4
          dc h'000F0FFF'    ;Row 5
          dc h'00F000F0'    ;Row 6
          dc h'OFFF000F'    ;Row 7 (bottom)

```

While QuickDraw's pen *always* draws using a pattern, you can make it draw with a solid color and, in effect, use no pattern at all. To do this, simply fill the PenPat table with the number of one specific color (e.g., \$0 for white).

In the super hi-res 640 mode, the PenPat table is also 256 bits long, but it defines colors for a block of 128 pixels (not 64, as in the 320 mode). This pixel block is 8 pixels high and 16 pixels wide.

Each entry in a 640 mode PenPat table is 2 bits long, because this mode can only display a pixel in one of four colors. (A discussion of colors is upcoming.) Here, black is numbered 0 and white is numbered 2.

**Figure 7-7**

Pen Mask

The pen mask (PenMask, in tool calls) lets you select which pixels in a pattern will be displayed when the pen draws. The mask is 64 bits long. In 320 mode, it has 1 bit for every pixel in the 8-by-8 pattern; in 640 mode, it has 1 bit for every *pair* of pixels in the 8-by-16 pattern. The rationale behind the 1-bit-per-pair layout relates to the way the 640 mode displays colors. More about that later.

Within the mask, each “1” bit permits the corresponding pixel (or pixel pair, in 640 mode) in the pattern to be displayed. Conversely, each “0” bit prevents its pattern counterpart from being displayed, and shows the background color instead. In short, the pen mask is a filter that selectively permits or blocks (1 or 0) the appearance of a pattern’s pixels.

Figure 7-8 shows a pen mask that blocks out every other pixel in a 320-mode pen pattern. In the pen mask shown here, the black boxes represent “1” (permit) bits and the white boxes represent “0” (block) bits.

You probably won’t want to apply a mask to very many patterns, and so you would fill it with 1’s, like this:

```
NoMask dc h'FF FF FF FF FF FF FF FF' ;No mask
```

Still, masks can be handy for using portions of a given pattern in several different drawing operations. For example, you may want to draw with the entire pattern at one place on the screen and with just the top half at another

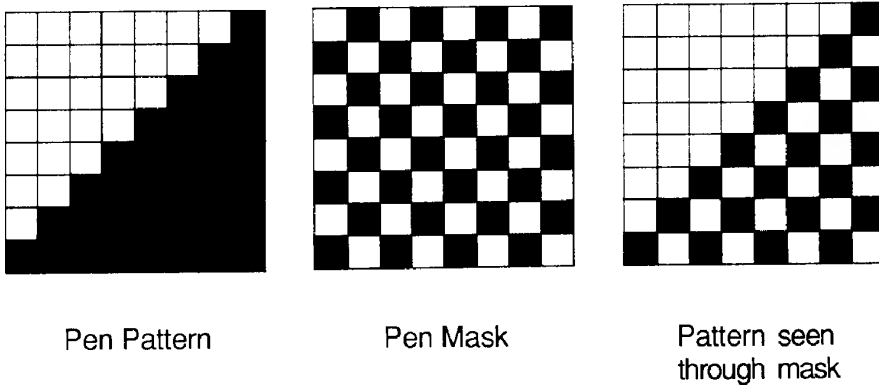


Figure 7-8

place. Preparing for this simply involves creating two masks (which is relatively easy), not two patterns (which is more difficult).

Pen State Tool Calls

Table 7-1 summarizes the tool calls you can use to read or set the pen state and its individual attributes. Note especially `PenNormal`, which sets the pen state (except for location) to QuickDraw's default values. `PenNormal` creates a 1-pixel pen that draws a solid white pattern with no mask. In the next section, I describe a few additional tool calls that govern the colors of pixels within the pen's pattern.

Colors

As mentioned earlier in this chapter, the two super hi-res graphics modes differ in the number of pixels and colors they display. One mode divides the screen into a pixel grid that is 200 rows high and 320 columns wide, and lets you display any pixel in 16 different colors. The other mode divides the screen into a grid of 200 rows and 640 columns, but lets you choose from only 4 pixel colors.

You're probably wondering why anyone would use a mode that offers only 4 colors. Actually, this mode is not as limiting as it sounds; you can get excellent color effects with it. Exactly *how* these effects are produced is not particularly intuitive, however, so I'll start by discussing the mode that

Table 7-1

Pen State Calls		
<code>__PenNormal</code>		Set the pen state to the normal values
Call with: <code>__PenNormal</code>		
Result: None		
Note: The standard pen state is <code>PenSize = 1,1</code> ; <code>PenMode = 0</code> (copy); <code>PenPat = black</code> ; <code>PenMask = 1</code> 's (no mask). The pen location is not changed.		
<code>__GetPenState</code>		Read the pen state
Call with: <code>PushLong PenState</code>	<code>;Pointer to PenState buffer</code>	
<code>__GetPenState</code>		
Result: None		
Note: <code>PenState</code> is:		
<code>ds 4</code>	<code>;Row and column of PenLoc</code>	
<code>ds 4</code>	<code>;Width and height of PenSize</code>	
<code>ds 2</code>	<code>;Pen mode</code>	
<code>ds 32</code>	<code>;Pen pattern</code>	
<code>ds 8</code>	<code>;Pen mask</code>	
<code>__SetPenState</code>		Set the pen state to the specified values
Call with: <code>PushLong PenState</code>	<code>;Pointer to PenState record</code>	
<code>__SetPenState</code>		
Result: None		
Note: <code>PenState</code> is:		
<code>dc i'Vert,Horiz'</code>	<code>;Row and column of PenLoc</code>	
<code>dc i'Width,Height'</code>	<code>;Width and height of PenSize</code>	
<code>dc i'PenMode'</code>	<code>;Pen mode</code>	
<code>dc h'PenPattern'</code>	<code>;Pen pattern</code>	
<code>dc h'Pen mask'</code>	<code>;Pen mask</code>	
Pen Location Calls		
<code>__GetPen</code>		Read the current pen location
Call with: <code>PushLong Point</code>	<code>;Pointer to point variable</code>	
<code>__GetPen</code>		
Result: None		
Note: The pen's coordinates are returned in <code>Point</code> , which should be defined as:		
<code>Point ds 4</code>		
GetPen will store the row number in the first 2 bytes and the column number in the last 2 bytes.		
<code>__MoveTo</code>		Move the pen to the specified point
Call with: <code>PushWord Horiz</code>	<code>;Column</code>	
<code>PushWord Vert</code>	<code>;Row</code>	
<code>__MoveTo</code>		
Result: None		
<code>__Move</code>		Move the pen by the specified displacements
Call with: <code>PushWord dHoriz</code>	<code>;Columns to move</code>	
<code>PushWord dVert</code>	<code>;Rows to move</code>	
<code>__Move</code>		
Result: None		

Table 7-1 (cont.)

Pen Size Calls		
<code>__GetPenSize</code>		Read the current pen size
Call with:	<code>PushLong <i>SizePtr</i></code> ;Pointer to <code>PenSize</code> loc.	
	<code>__GetPenSize</code>	
Result:	None	
Note:	<code>SizePtr</code> points to:	
	<code>ds 2</code> ;Width	
	<code>ds 2</code> ;Height	
<code>__SetPenSize</code>		Set the pen size
Call with:	<code>PushWord <i>Width</i></code>	
	<code>PushWord <i>Height</i></code>	
	<code>__SetPenSize</code>	
Result:	None	
Pen Pattern Calls		
<code>__GetPenPat</code>		Read the current pen pattern
Call with:	<code>PushLong <i>PatternBuf</i></code> ;Pointer to pattern buffer	
	<code>__GetPenPat</code>	
Result:	None	
Note:	<code>PatternBuf</code> is:	
	<code>ds 32</code>	
<code>__SetPenPat</code>		Set the pen pattern to the specified values
Call with:	<code>PushLong <i>PatternPtr</i></code> ;Pointer to pattern	
	<code>__SetPenPat</code>	
Result:	None	
Note:	<code>PatternPtr</code> points to:	
	<code>dc h'<i>Pattern</i>'</code>	
Pen Mask Calls		
<code>__GetPenMask</code>		Read the current pen mask
Call with:	<code>PushLong <i>MaskBuf</i></code> ;Pointer to mask buffer	
	<code>__GetPenMask</code>	
Result:	None	
Note:	<code>MaskBuf</code> is:	
	<code>ds 8</code>	
<code>__SetPenMask</code>		Set the pen mask to the specified values
Call with:	<code>PushLong <i>MaskPtr</i></code> ;Pointer to mask	
	<code>__SetPenMask</code>	
Result:	None	
Note:	<code>MaskPtr</code> points to:	
	<code>dc h'<i>Mask</i>'</code>	

displays 200-by-300 pixels in 16 colors — the so-called 320 mode — which is easier to understand.

320 Mode Colors

Recall that a 320 mode pen pattern consists of 4-bit values (nibbles) that select the color for each pixel in an 8-by-8 array. Each value is actually an index that QuickDraw uses to obtain color information from a 16-entry *color table*. The Apple IIGS provides 16 different color tables for 320 mode (and you can create others of your own), but most people use the default, or *standard*, color table.

Table 7-2 shows the standard color table for 320 mode (the one for 640 mode is different). Here, the Pixel Value column lists the hexadecimal digit that you would enter in your pen pattern and the Master Color Value column lists the 3-digit number that QuickDraw sends to the computer's color-generating circuitry.

The master color values may appear rather arbitrary, but by examining them closely you can appreciate how they relate to the colors they produce. For starters, note that black has the value 000, while white has the value FFF. This is appropriate, because black and white are, in fact, opposites. Black is the total absence of color, while white is a combination of the primary colors, red, green, and blue.

You can better understand the makeup of the master color values by examining the primary colors. Red has the value D00; its first digit is high-numbered digit (D) and the other digits are 0. Similarly, green (0E0) has a high-numbered second digit and 0 in the other positions, while blue (00F) has a high-numbered third digit and 0 in the other positions.

Table 7-2

Pixel Value	Color	Master Color Value
0 (\$0)	Black	000
1 (\$1)	Dark Gray	777
2 (\$2)	Brown	841
3 (\$3)	Purple	72C
4 (\$4)	Blue	00F
5 (\$5)	Dark Green	080
6 (\$6)	Orange	F70
7 (\$7)	Red	D00
8 (\$8)	Flesh	FA9
9 (\$9)	Yellow	FF0
10 (\$A)	Green	0E0
11 (\$B)	Light Blue	4DF
12 (\$C)	Lilac	DAF
13 (\$D)	Periwinkle Blue	78F
14 (\$E)	Light Gray	CCC
15 (\$F)	White	FFF

The conclusion is, then, *each digit in a master color value indicates the intensity of a primary color*. The first digit indicates how much red the pixel has, the middle digit how much green, and the third how much blue. For example, the value for yellow, FF0, tells you that it contains equal parts of red and green (with both at full intensity), but no blue.

640 Mode Colors

Recall that in 640 mode, the pixel values are only 2 bits long, allowing you to select from four colors (numbered 00, 01, 10, and 11). That being the case, you would expect the 640 mode's color table to be four entries long. Fortunately, it isn't; it has 16 entries, the same as in 320 mode.

You're probably wondering how a 2-bit number can be used to obtain color information from a 16-entry table. The fact is that the table consists of four *minipalettes*, each containing four master color values (see Table 7-3), from which a 2-bit pixel value selects a color. (Note that Table 7-3 shows the default color table for 640 mode. As with 320 mode, the Apple IIGS provides 15 additional tables. They are discussed at the end of this chapter.)

From which minipalette does QuickDraw obtain the color value for a pixel? When displaying graphics images, QuickDraw assigns 4 horizontally

Table 7-3

Pixel Value	Color	Master Color Value	
0	Black	000	<i>Minipalette 0</i>
1	Red	F00	
2	Green	0F0	
3	White	FFF	
0	Black	000	<i>Minipalette 1</i>
1	Blue	00F	
2	Yellow	FF0	
3	White	FFF	
0	Black	000	<i>Minipalette 2</i>
1	Red	F00	
2	Green	0F0	
3	White	FFF	
0	Black	000	<i>Minipalette 3</i>
1	Blue	00F	
2	Yellow	FF0	
3	White	FFF	

adjacent pixels to 4 consecutive minipalettes in the table. However, it does *not* access the minipalettes in the 0-1-2-3 order one might expect. Instead, *Quickdraw reads the color value of the 4 pixels from minipalettes 2, 3, 0, and 1, respectively*. It starts halfway down the table and “wraps around” to the beginning!

How can one produce worthwhile colored graphics using the 640 mode color table? After all, minipalettes 0 and 2 are identical, as are 1 and 3. Moreover, black and white appear in every minipalette, which doesn’t leave many entries open for additional colors. In short, QuickDraw provides only six colors: black, white, red, green, blue, and yellow. That’s not much to work with . . . or is it?

You *can*, in fact, produce dazzling color graphics in 640 mode by employing a display technique called *dithering*.

Dithering in 640 Mode

Early in the development of the IIGS, Apple’s engineers created a simple demonstration program that drew colored lines on the screen. It ran in 320 mode and generated a color for each line as a random 4-bit number between 0 and 15 (for the 16 entries in the color table). Rather than rewrite the entire program for the 640 mode, someone changed only the variable that selects the mode, then assembled the program and ran it.

Running in 640 mode, the program produced effects the engineers had not expected. Some lines showed one of the regular 640 mode colors, but others showed a shade produced when two of the regular colors were combined! In fact, there were *16* different shades in all.

Here’s what had happened. Aside from changing the mode select variable, the engineer who modified the program made no special provision for the fact that it was to run in 640 mode. That is, he or she did not allow for the difference in pen pattern formats between the modes. (Remember, 320 mode patterns consist of 4-bit values, but 640 patterns use 2-bit values.) As a result, while the 320 mode version of the program produced lines of one specific color from the table, the 640 mode version produced lines that combined two color values.

For example, if the program were to draw a line with color 6 (binary 0110) in 320 mode, it would use the seventh entry in the 320 mode color table — orange. However, receiving a 6 in 640 mode made the program break the 4-bit number into two parts and handle each part separately. Here, the program used the individual parts (01 and 10, respectively) to obtain color values from two different minipalettes. In doing this, it displayed the

first pixel as red and the second as yellow. Because pixels are extremely small on a 640 mode screen, when the red and yellow pixels were shown side by side, they appeared as the combination color, *orange*!

The important discovery that Apple's engineers made with their early 640-mode program was this: *when 2 pixels of different colors appear side by side, the eye sees them as a combination color, or shade*. This effect is called dithering.

By carefully selecting the pen pattern values your program draws with, you can produce the 16 shades that are available with dithering. Figure 7-9 summarizes these shades and the hex digit you must enter in a pen pattern to select each one. For example, to draw in blue, fill your pen pattern with the value 1; to draw in pastel green, fill it with B.

Tool Calls for Color

As you now know, the color or colors with which the pen draws depends on the values in the current pen pattern (PenPat). Table 7-1, shown earlier, lists three tool calls (PenNormal, SetPenState, and SetPenPat) that change the pen pattern, and two others (GetPenState and GetPenPat) that read the contents of the current pattern. Table 7-4 lists additional tool calls that involve the pen pattern, and thus, the color with which QuickDraw draws. This table also has calls that work with the screen's background pattern (or color).

When drawing with solid colors in 320-mode, you can use `SetSolidPenPat` and `SetSolidBackPat` to select your colors, and disregard the `SetPenPat` (described earlier) and `SetBackPat` calls. Here, the `ColorNum` input lets you specify any of the 16 colors (0 through 15) in the 320-mode color table.

SetSolidPenPat and **SetSolidBackPat** will also do the job in 640 mode, *provided* you only want one of the first four colors in the table — black, red, green, or white. To produce any other color, you must set up a dithered pattern (using the values in Figure 7-9) and call **SetPenPat** or **SetBackPat** to make QuickDraw use it. For example, to draw in blue on a 640-mode screen, you must first set the color with the following kind of sequence:

```

                                Pushlong #BluePenPat      ; Pens should
                                _SetPenPat                ; draw in blue
                                . .
                                . .
BluePenPat    dch'11 11 11 11 11 11 11 11 11 11'      ;Dithered
              dch'11 11 11 11 11 11 11 11 11 11'      ;blue
              dch'11 11 11 11 11 11 11 11 11 11'      ;pattern
              dch'11 11 11 11 11 11 11 11 11 11'
```


	Black	Blue	Yellow	White
Black	Black 0	Blue 1	Gold 2	Grey 3
Red	Red 4	Violet 5	Orange 6	Pink 7
Green	Green 8	Turquoise 9	Chartreuse A	Pastel Green B
White	Gray C	Pastel Blue D	Yellow E	White F

Figure 7-9

Drawing Lines, Rectangles, and Polygons

Now that you know how to set up the pen and work with its attributes, it's time to find out how to draw with it. QuickDraw has tool calls that let you draw lines, rectangles (with either square or rounded corners), polygons, ovals (or circles), and arcs. It also lets you draw *regions*: areas that contain several shapes. This section covers lines, rectangles, and polygons. Ovals, arcs, and regions are described in the next section.

QuickDraw always starts drawing at the current pen location and uses the current pen size, mode, pattern (with some exceptions), and mask. Hence, before drawing something, you must ensure that the pen has the attributes you want.

Table 7-4

Pen Pattern Calls	
__SetSolidPenPat	Set the pen pattern to a solid color
Call with: PushWord <i>ColorNum</i> ;Color number	
__SetSolidPenPat	
Result: None	
Note: QuickDraw only uses the appropriate number of bits in <i>ColorNum</i> . In 320 mode, it uses 4 bits, so <i>ColorNum</i> can range from 0 to 15. In 640 mode, it uses 2 bits, so <i>ColorNum</i> can be 0, 1, 2, or 3. In the default palette, this is black, red, green, and white, respectively.	
__SolidPattern	Set the specified pen pattern to a solid color
Call with: PushLong <i>PatternPtr</i> ;Pointer to pattern	
PushWord <i>ColorNum</i> ;Color number	
__SolidPattern	
Result: None	
Note: See note for SetSolidPenPat.	
Background Pattern Calls	
__GetBackPat	Read the background pattern
Call with: PushLong <i>PatternPtr</i> ;Pointer to pattern buffer	
__GetBackPat	
Result: None	
Note: <i>PatternPtr</i> points to: ds 32	
__SetBackPat	Set the background to the specified pattern
Call with: PushLong <i>PatternPtr</i> ;Pointer to pattern	
__SetBackPat	
Result: None	
Note: <i>PatternPtr</i> points to: dc h' <i>Pattern</i> '	
__ClearScreen	Clear the screen to the specified color
Call with: PushWord <i>ColorWord</i>	
__ClearScreen	
Result: None	
Note: To obtain a solid color on the screen, all 4 hex digits in <i>ColorWord</i> must be identical. For example, to set a 320 mode screen to light blue, enter PushWord #5BBB5B .	

Lines

A line is the easiest object to draw. As Table 7-5 shows, there are only two line-drawing calls: **LineTo** and **Line**. They're similar, but with **LineTo** you specify the end point of the line, whereas with **Line** you specify displacements relative to the starting point.

Table 7-5

<u>LineTo</u>		Draw a line to the specified point
Call with:	PushWord <i>Horiz</i> ;Column PushWord <i>Vert</i> ;Row <u>LineTo</u>	
Result:	None	
<u>Line</u>		Draw a line to the point specified by displacements
Call with:	PushWord <i>dHoriz</i> ;End is dHoriz columns PushWord <i>dVert</i> ; and dVert rows away <u>Line</u>	
Result:	None	

To draw a line, move the pen to its starting point and give a `LineTo` call. For example, to connect points (10,20) and (30,40), enter:

```

PushWord #20      ;Specify the starting column
PushWord #10      ; and row positions
_MoveTo          ;Move the pen there
PushWord #40      ;Specify the ending column
PushWord #30      ; and row positions
_LineTo

```

Note that a `Line` call with 20 for both the horizontal and vertical displacement would do the same thing as the preceding `LineTo` call. `Line`'s parameters are signed values, so you can specify negative as well as positive displacements. For example, a vertical displacement of -10 sets the end point 10 rows above the starting point.

Rectangles

QuickDraw has four tool calls for drawing rectangles (see table 7-6). For each, you must supply a pointer to the rectangle's top left-hand corner and bottom right-hand corner coordinates. This is a data structure of the form:

```

dc i' V1, H1'      ;Row, column of top left-hand corner
dc i' V2, H2'      ;Row, column of bottom right-hand corner

```

The first call, `FrameRect`, draws the boundary or *frame* of a rectangle and shows the inside in the background pattern. For example, the following

Table 7-6

<code>__FrameRect</code>	Draw the boundary of the specified rectangle
Call with: <code>PushLong RectPtr</code>	;Pointer to rectangle definition
	<code>__FrameRect</code>
Result:	None
Note:	<code>RectPtr</code> points to:
	<code>dc i'V1,H1'</code> ;Row, column of top left-hand corner
	<code>dc i'V2,H2'</code> ;Row, column of bottom right-hand corner
<code>__PaintRect</code>	Fill the interior of a rectangle with the current pen pattern
Call with: <code>PushLong RectPtr</code>	;Pointer to rectangle definition
	<code>__PaintRect</code>
Result:	None
Note:	See <code>__FrameRect</code> for the rectangle definition.
<code>__FillRect</code>	Fill the interior of a rectangle with the specified pen pattern
Call with: <code>PushLong RectPtr</code>	;Pointer to rectangle definition
	<code>PushLong PatternPtr</code> ;Pointer to pattern
Call with: <code>__FillRect</code>	
Result:	None
Note:	See <code>__FrameRect</code> for the rectangle definition. <code>PatternPtr</code> points to:
	<code>dc h'Pattern'</code>
<code>__EraseRect</code>	Fill the interior of a rectangle with the background color
Call with: <code>PushLong RectPtr</code>	;Pointer to rectangle definition
	<code>__EraseRect</code>
Result:	None
Note:	See <code>__FrameRect</code> for the rectangle definition.

draws the frame of a rectangle whose top left-hand corner is at (10,20) and bottom right-hand corner is at (40,60):

```

                PushLong #Rect1    ;Pointer to rectangle
                __FrameRect        ; coordinates
                . .
                . .
Rect1    dc i'10,20'                ;Top left-hand corner
                dc i'40,60'          ;Bottom right-hand corner

```

You can use two other calls to draw solid rectangles. `PaintRect` draws a solid rectangle using the current pen pattern, while `FillRect` draws one using a pen pattern that you specify. Finally, the `EraseRect` call removes a rectangle from the screen by replacing it with the background pattern.

Be aware, incidentally, that any solid rectangle (or other shape) you draw replaces whatever is previously displayed within its boundary. Thus, to draw one solid rectangle inside another, always draw the outer rectangle first. If you draw the inner one first, it won't appear on the screen; the outer rectangle will obliterate it.

By now, you're probably tired of reading about tool calls and want to see some of them used in an actual program. That's next on the agenda.

A Program that Draws Rectangles

This section presents a program that draws four rectangles. Before looking at the actual listing, let's see what the program does and what it produces on the screen.

When I first started to think about this example program, I decided that it should include a mixture of tool calls that illustrate not only the use of `FrameRect` and `PaintRect`, but show the use of some pen-related tool calls as well. To start working on the program, I drew the rectangles on a piece of graph paper and named them `Rect1`, 2, 3, and 4.

Figure 7-10 is similar to (but much neater than) my original drawing. Note that `Rect1` and `Rect3` are frames or boundaries, while `Rect2` and `Rect4` are solid. The black borders that enclose `Rect1` represent the default background color, black.

Here, then, are the specifications for my four rectangles:

- *Rect1* is a frame whose top left and bottom right corners are at (70,90) and (155,310). It is drawn with the default pen size (1,1) and a solid pen pattern of color 5. This dither of red (binary 01) and blue (01) produces violet.
- *Rect2* is a solid rectangle whose top left and bottom right corners are at (75,100) and (150,300). It is drawn with the default pen size (1,1) and a solid pattern of color \$F, white.
- *Rect3* is a frame whose top left and bottom right corners are at (60,75) and (165,320). It is drawn with a pen size of (5,5) and a solid pattern of color 2. This produces a dither of black (00) and green (10), which appears as green.
- *Rect4* is a solid rectangle whose top left and bottom right corners are at (60,375) and (165,620). It is drawn with a pen size of (5,5) and a solid pen pattern of color 5. As with `Rect1`, this dither of red (01) and blue (01) produces violet.

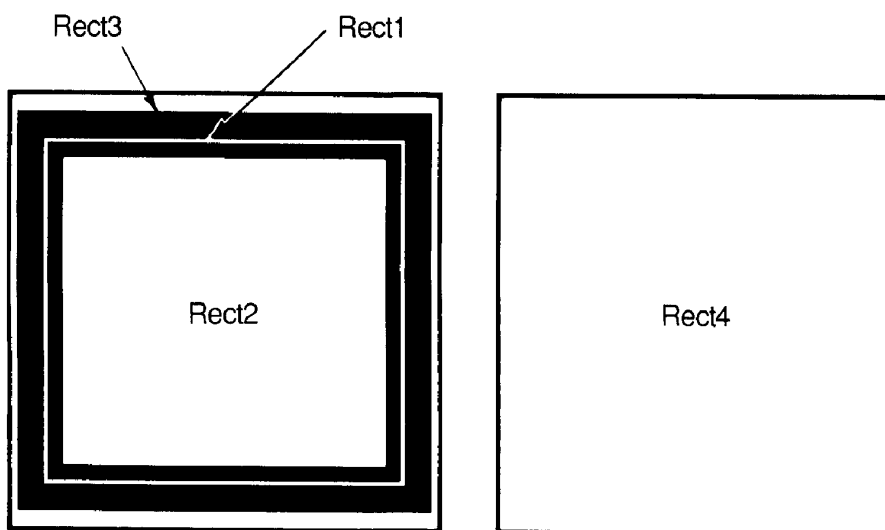


Figure 7-10

My final consideration was the mode. Drawing in 320 mode would have made my job somewhat easier, but I chose 640 mode instead, to illustrate the dithering effect.

Example 7-1 lists my final rectangle-drawing program, RECTS, which has two data segments and four code segments. One data segment, Global-Data, contains variables used throughout the program; the other, RectData, contains the rectangle coordinates and a pen state record.

The first code segment, Rects, simply keeps a global equate separate from other code. The DoIt segment is the program's "main lobby." It calls the InitStuff subroutine to start up the tools, DrawRects to draw the rectangles, and EventLoop to wait for the user's signal to quit. Finally, tool calls at the end of DoIt shut down the tools and exit.

Note that the user interface segment, EventLoop, contains three instructions that keep the rectangles on the screen until the user presses a key. This is an Apple II way of waiting for a key. You can accomplish the same thing with tool calls, but since I have not yet introduced those calls, I can't, in fairness, use them.

Polygons

Polygons are shapes that have three or more sides, such as triangles, hexagons, and octagons. To display a polygon, you must define its shape, then

Example 7-1

; RECTS draws four rectangles in 640 mode.

```

        absaddr on
        MCOPY Rects.macros

Rects      START
           using GlobalData

;-----
;
; Global equate used throughout the program.

ScreenMode    gequ $80                ;640 mode, no fill

           phk                        ;Set data bank to program
           plb                        ; bank to allow absolute addressing

           jsr InitStuff              ;Initialize everything
           bcs AllDone               ;Quit if initialization fails

           jsr DrawRects              ;Draw the rectangles

           jsr EventLoop              ;Wait for user to press a key

AllDone       anop                    ;All is done, shut down
               _QDShutDown            ;QuickDraw II
               _MTShutDown            ;Miscellaneous Tools

           PushWord MyID              ;Discard the program's handle
               _DisposeAll

           PushWord MyID              ;Memory manager
               _MMShutdown            ;Tool Locator
               _TLShutdown

               _Quit QuitParams       ;Do a ProDOS Quit call
               brk $F0                ;If it fails, break

           END

;*****
;
; Global Data
;
;*****

GlobalData    DATA

QuitParams    dc i4'0'                ;Return to caller
               dc i'$4000'            ;Make program restartable in memory

MyID          ds 2                    ;This will hold the program's i.d.

           END

```

216 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

```

;*****
;
; InitStuff
;
; Initializes tool sets and gets space in bank 0 for use as
; direct page by tools that need it.
;
;*****

InitStuff      START
               using GlobalData

; Initialize the Tool Locator, Memory Manager, and Miscellaneous
; Tools.

               _TLStartup

               PushWord #0
               _MMStartup

               pla                               ;Memory Manager returns program's ID
               sta MyID

               _MTStartup
               ldx #3
               jsr PrepareToDie

; Get some space for the zero page we need. QuickDraw needs
; three pages.

               PushLong #0                      ;Space for handle
               PushLong #$300                   ;Three pages for QuickDraw
               PushWord MyID                    ;Owner
               PushWord #$C005                  ;Locked, fixed, fixed bank
               PushLong #0                      ;Location
               _NewHandle
               ldx #$FF
               jsr PrepareToDie

               pla                               ;Read handle
               sta 0                             ; and store in direct page
               pla
               sta 2

               lda [0]                          ;Get DP location from handle

; Initialize QuickDraw

               pha                               ;Use dp obtained from handle
               PushWord #ScreenMode             ;Mode = 640
               PushWord #0                      ;Use entire screen
               PushWord MyID
               _QDStartup
               ldx #4
               jsr PrepareToDie

               clc                               ;Clear the carry flag
               rts                             ; and return

END

```



```

;*****
;
; Draw the rectangles
;
;*****

DrawRects START
    using RectData

    PushWord #5                ;Set the pen color
    _SetSolidPenPat

    PushLong #Rect1            ; and draw the Rect1 frame
    _FrameRect

    PushWord #$F               ;Now switch to white
    _SetSolidPenPat

    PushLong #Rect2            ; and paint Rect2
    _PaintRect

; Change pen attributes one at a time

    PushWord #5                ;Size is 5,5
    PushWord #5
    _SetPenSize

    PushWord #0                ;Regular mode
    _SetPenMode

    PushWord #2                ;Pattern
    _SetSolidPenPat

    PushLong #PenMask          ;Mask
    _SetPenMask

; Let's see what the frame looks like

    PushLong #Rect3            ;Draw Rect3 frame
    _FrameRect

; Now an example of how to set up everything at once

    PushLong #PenRec
    _SetPenState

    PushLong #Rect4            ;Paint Rect4
    _PaintRect

    rts
    END

;*****
;
; Event Loop
;
; Keep the rectangles on the screen until user presses a key.
;
;*****

```

218 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

EventLoop START

```
WaitHere        lda $E0C000
                 bpl WaitHere
                 lda $E0C010
                 rts
```

END

```
;*****
;
; Rectangle Data
;
; Coordinates of the rectangles and a pen state record.
;
;*****
```

RectData DATA

```
Rect1    dc i'70'                                ;Top left corner is at
           dc i'90'                                ; 70,90
           dc i'155'                              ;Bottom right corner is at
           dc i'310'                              ; 155,310

Rect2    dc i'75'                                ;Top left corner is at
           dc i'100'                              ; 75,100
           dc i'150'                              ;Bottom right corner is at
           dc i'300'                              ; 150,300

Rect3    dc i'60'                                ;Top left corner is at
           dc i'75'                                ; 60,75
           dc i'165'                              ;Bottom right corner is at
           dc i'320'                              ; 165,320

Rect4    dc i'60'                                ;Top left corner is at
           dc i'375'                              ; 60,375
           dc i'165'                              ;Bottom right corner is at
           dc i'620'                              ; 165,620
```

```
PenRec    anop
PenLoc    dc i'0,0'                                ;Pen location
PenSize   dc i'5,5'                                ;Pen size
PenMode   dc i'0'                                  ;Pen mode
PenPat    dc h'55 55 55 55 55 55 55 55'        ;Pen Pattern
           dc h'55 55 55 55 55 55 55 55'
           dc h'55 55 55 55 55 55 55 55'
           dc h'55 55 55 55 55 55 55 55'
PenMask   dc h'FF FF FF FF FF FF FF FF'        ;No mask
```

END

```
*****
*
* Prepare To Die
*
* Dies if carry is set.
*
* Error number to display is in X register.
* Assumes that Miscellaneous Tools is active.
*
*****
```

```

PrepareToDie  START
               bcs RealDeath      ; Carry = 1?
               rts                 ; No. Return to caller

RealDeath     phx                  ; Yes. Goodbye, program.
               PushLong #DeathMsg
               _SysFailMgr

DeathMsg      str 'Could not handle error '

               END

```

draw it. Defining the shape involves entering a series of MoveTo and LineTo calls that specify how the pen should move. To tell QuickDraw how far the definition extends, you must enter an OpenPoly call at the beginning of it and a ClosePoly call at the end. Table 7-7 summarizes OpenPoly and ClosePoly and the calls that actually draw or erase a polygon.

Note that just as the Memory Manager assigns a handle to your program when you call NewHandle, QuickDraw assigns a handle to a polygon definition when you call OpenPoly. This handle is what you need to draw or erase the polygon. You must also discard the handle (using DisposHandle) at the end of your program, just as you must discard the program's handle.

Example 7-2 lists a program called POLY that paints a right triangle (one that contains a 90-degree angle) in 640 mode at the bottom of the screen. Like the earlier RECTS program, POLY includes Apple II-style instructions that keep the figure on the screen until the user presses a key.

Drawing Other Shapes

QuickDraw can also draw ovals, arcs, regions, and rectangles with round corners (*roundrects*). Most of the tool calls for these shapes are RAM-based, as opposed to the calls that draw lines, rectangles, and polygons, which are in ROM. To run a program that calls a RAM-based tool, you must start the computer by booting up the System Disk (that loads the RAM-based tools in memory).

Recall from Chapter 6 that there is a set of “MountBookDisk calls that prompt the user to insert the System Disk if he or she has violated this rule. The program model includes these calls and so should your programs.

This section describes only how to draw ovals (including circles) and regions. Arcs and roundrects are generally less useful for application programs, but if you need them, refer to Apple's documentation.

Table 7-7

Definition Calls		
__OpenPoly		Get a handle for a polygon data structure
Call with:	PushLong # __OpenPoly	;Space for result (handle)
Result:	Handle (long word)	
Note:	OpenPoly begins the definition of a polygon. (Define the polygon with MoveTo and LineTo calls.) ClosePoly ends it.	
__ClosePoly		End the current polygon definition
Call with:	__ClosePoly	
Result:	None	
Drawing Calls		
__FramePoly		Draw the boundary of the specified polygon
Call with:	PushLong <i>PolyHandle</i> __FramePoly	;Polygon handle
Result:	None	
__PaintPoly		Fill the interior of a polygon with the current pen pattern
Call with:	PushLong <i>PolyHandle</i> __PaintPoly	;Polygon handle
Result:	None	
__FillPoly		Fill the interior of a polygon with the specified pen pattern
Call with:	PushLong <i>PolyHandle</i> PushLong <i>PatternPtr</i>	;Polygon handle ;Pointer to pattern
Result:	None	
__ErasePoly		Erase the specified polygon
Call with:	PushLong <i>PolyHandle</i> __ErasePoly	;Polygon handle
Result:	None	

Ovals

Anyone who has taken geometry class might expect an oval-drawing operation to involve arcs and angles and the like. Fortunately, that's not the case with QuickDraw. As Table 7-8 shows, to draw an oval, you simply specify the corner points of the *rectangle* in which the oval is to be inscribed.

For example, to produce an oval that is 30 pixels high and 15 pixels wide, you would specify a rectangle of that size when you make your oval-drawing call. You can also use the oval calls to draw *circles*, by making your rectangle square.

Example 7-2

; POLY draws a triangle in 640 mode.

```

        absaddr on
        MCOPY Poly.macros

Poly          START
              using GlobalData

;-----
;
; Global equate used throughout the program.

ScreenMode    gequ $80                ;640 mode, no fill

              phk                      ;Set data bank to program
              plb                      ; bank to allow absolute addressing

              jsr InitStuff            ;Initialize everything
              bcs AllDone              ;Quit if initialization fails

              jsr DrawPoly             ;Draw the triangle

              jsr EventLoop            ;Wait for user to press a key

AllDone       anop                    ;All is done, shut down
              _QDShutDown              ;QuickDraw II
              _MTShutDown              ;Miscellaneous Tools

              PushWord MyID            ;Discard the program's handles
              _DisposeAll

              PushWord MyID            ;Memory Manager
              _MMShutDown              ;Tool Locator
              _TLShutDown

              _Quit QuitParams         ;Do a ProDOS Quit call
              brk $F0                  ;If it fails, break

              END

;*****
;
; Global Data
;
;*****

GlobalData    DATA

TriangleH     ds 4                    ;This will hold triangle's handle

QuitParams    dc i4'0'                ;Return to caller
              dc i'$4000'             ;Make program restartable in memory

MyID          ds 2                    ;This will hold the program's i.d.

              END

```



```

;*****
;
; Define the polygon, then draw it
;
;*****

DrawPoly  START
          using GlobalData

; This is the polygon's definition

          PushLong #0                ;Space for result (handle)
          _OpenPoly                  ;Start the definition
          pla                        ;Read the handle into TriangleH
          sta TriangleH
          pla
          sta TriangleH+2

          PushWord #100              ;Move pen to starting point
          PushWord #100              ; (100,100)
          _MoveTo

          PushWord #100              ;Draw a vertical line
          PushWord #200              ; from (100,100) to (200,100)
          _LineTo

          PushWord #250              ;Draw a horizontal line
          PushWord #200              ; from (200,100) to (200,250)
          _LineTo

          PushWord #100              ;Draw a line back to the
          PushWord #100              ; starting point
          _LineTo

          _ClosePoly                ;End the polygon definition

; Set up the pen and draw the triangle

          PushWord #5                ;Set the pen color
          _SetSolidPenPat

          PushLong TriangleH         ; and paint the triangle
          _PaintPoly

          rts
          END

;*****
;
; Event Loop
;
; Keep the triangle on the screen until user presses a key.
;
;*****

EventLoop  START

WaitHere   lda $E0C000
           bpl WaitHere

```

```

        lda $E0C010
        rts

END

*****
*
* Prepare To Die
*
* Dies if carry is set.
*
* Error number to display is in X register.
* Assumes that Miscellaneous Tools is active.
*
*****

PrepareToDie  START
               bcs RealDeath      ;Carry = 1?
               rts                ; No. Return to caller

RealDeath     phx                  ; Yes. Goodbye, program.
               PushLong #DeathMsg
               __SysFailMgr

DeathMsg      str 'Could not handle error '

END

```

Table 7-8

<u>__FrameOval</u>	Draw the boundary of an oval inscribed in the specified rectangle
Call with: PushLong <i>RectPtr</i>	;Pointer to rectangle definition
	<u>__FrameOval</u>
Result:	None
Note:	RectPtr points to:
	dc i' V1,H1' ;Row, column of top left-hand corner
	dc i' V1,H2' ;Row, column of bottom right-hand corner
<u>__PaintOval</u>	Fill the interior of an oval with the current pen pattern
Call with: PushLong <i>RectPtr</i>	;Pointer to rectangle definition
	<u>__PaintOval</u>
Result:	None
Note:	See <u>__FrameOval</u> for the rectangle definition.
<u>__FillOval</u>	Fill the interior of an oval with the specified pen pattern
Call with: PushLong <i>RectPtr</i>	;Pointer to rectangle definition
	PushLong <i>PatternPtr</i> ;Pointer to pattern
	<u>__FillOval</u>
Result:	None
Note:	See <u>__FrameOval</u> for the rectangle definition. PatternPtr points to:
	dc h' <i>Pattern</i> '

Table 7-8 (cont.)

<code>__EraseOval</code>	Fill the interior of an oval with the background color
Call with: <code>PushLong RectPtr</code>	;Pointer to rectangle definition
<code>__EraseOval</code>	
Result: None	
Note: See <code>__FrameOval</code>	for the rectangle definition.

Regions

A region is a rectangular area that encloses one or more shapes. Once you have defined the shapes in the region, you can make QuickDraw display all of them at once by telling it to draw the region.

Defining a region is similar to defining a polygon; that is, you begin with an `OpenRgn` call, enter the drawing calls that define the shapes in the region, and end the definition with a `CloseRgn` call. However, because a region can hold several shapes, you must allocate space in memory for it with a `NewRgn` call. (It is `NewRgn`, not `OpenRgn`, that returns the region's handle.) The `CloseRgn` call at the end of the definition creates the region and frees the memory it uses. In summary, then, here is the procedure to create a region:

1. Call `NewRgn` to allocate memory for the region. `NewRgn` returns the region's handle.
2. Call `OpenRgn` to start defining the region.
3. Draw the shapes in the region. The legal calls here are `MoveTo`, `Line`, `LineTo`, `FrameRect`, `FrameOval`, `FrameRRect` (frame a roundrect), `FramePoly`, and `FrameRgn`.
4. Call `CloseRgn` to end the definition and free the allocated memory.
5. Draw the region with a `FrameRgn` or `PaintRgn` call.

Table 7-9 summarizes the tool calls most commonly used for regions.

Suppose, for example, you want to display a red clown-style bow tie (two triangles with a circular "knot" between them) that is 30 pixels high and 100 pixels wide. If you want the top left-hand point of the tie to appear at (30,30), Figure 7-11 shows the coordinates your program needs to draw it.

To produce the bow tie, the program must set up a region for it, make

Table 7-9

Definition Calls		
<hr/>		
<code>__NewRgn</code>		Allocate space for a new structure
Call with:	<code>PushLong #0</code>	;Space for result (handle)
	<code>__NewRgn</code>	
Result:	Handle (longword)	
Note:	CloseRgn frees the allocated space.	
<code>__OpenRgn</code>		Start defining a region
Call with:	<code>OpenRgn</code>	
Result:	None	
Note:	CloseRgn ends the region definition.	
<code>__CloseRgn</code>		End the current region definition
Call with:	<code>PushLong RgnHandle</code>	;Region handle
	<code>__CloseRgn</code>	
Result:	None	

Drawing Calls		
<code>__FrameRgn</code>		Draw the boundary of the specified region
Call with:	<code>PushLong RgnHandle</code>	;Region handle
	<code>__FrameRgn</code>	
Result:	None	
<code>__PaintRgn</code>		Fill the interior of a region with the current pen pattern
Call with:	<code>PushLong RgnHandle</code>	;Region handle
	<code>__PaintRgn</code>	
Result:	None	
<code>__FillRgn</code>		Fill the interior of a region with the specified pen pattern
Call with:	<code>PushLong RgnHandle</code>	;Region handle
	<code>PushLong PatternPtr</code>	;Pointer to pattern
	<code>__FillRgn</code>	
Result:	None	
<code>__EraseRgn</code>		Erase the specified region
Call with:	<code>PushLong RgnHandle</code>	;Region handle
	<code>__EraseRgn</code>	
Result:	None	

LineTo calls to define the triangular bows and a FrameOval call to define the knot, and give a region-drawing call make QuickDraw draw the region. Example 7-3 lists the program, called BOWTIE. Note that I had to load QuickDraw's RAM-based tools from disk (with LoadOneTool), to obtain the FrameOval call. If the region had no oval, that wouldn't have been necessary.

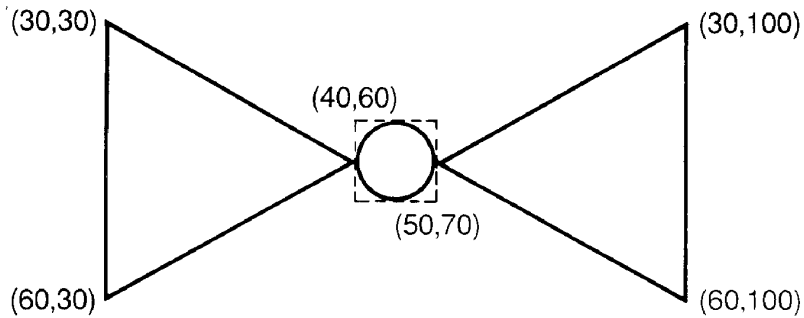


Figure 7-11

Example 7-3

; BOWTIE draws a region that defines a bowtie shape.

```
absaddr on
MCOPY Bowtie.macros
```

```
Bowtie      START
            using GlobalData
```

```
;-----
;
```

; Global equate used throughout the program.

```
ScreenMode    gequ $80                ;640 mode, no fill

phk            ;Set data bank to program
plb            ; bank to allow absolute addressing

jsr InitStuff    ;Initialize everything
bcs AllDone      ;Quit if initialization fails

jsr DrawTie      ;Draw the bowtie

jsr EventLoop    ;Wait for user to press a key

AllDone        anop                    ;All is done, shut down
                _QDShutDown            ;QuickDraw II
                _MTShutDown            ;Miscellaneous Tools

                PushWord MyID           ;Discard the program's handles
                _DisposeAll

                PushWord MyID           ;Memory Manager
                _MMShutDown            ;Tool Locator
                _TLShutDown

                _Quit QuitParams        ;Do a ProDOS Quit call
                brk $F0                ;If it fails, break

END
```

228 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

```
;*****
;
; Global Data
;
;*****

GlobalData    DATA

TieHandle     ds 4                ;This will hold bowtie's handle

KnotDef       dc i'40,60'        ;Rectangle that encloses knot
              dc i'50,70'

QuitParams    dc i4'0'           ;Return to caller
              dc i'$4000'        ;Make program restartable in memory

MyID          ds 2                ;This will hold the program's i.d.

VolNotFound   equ $45            ;ProDOS error

END

;*****
;
; InitStuff
;
; Initializes tool sets and gets space in bank 0 for use as
; direct page by tools that need it.
;
;*****

InitStuff     START
              using GlobalData

; Initialize the Tool Locator, Memory Manager, and Miscellaneous
; Tools.

        _TLStartup

        PushWord #0
        _MMStartup

        pla                                ;Memory Manager returns program's ID
        sta MyID

        _MTStartup
        ldx #3
        jsr PrepareToDie

; Get some space for the direct page we need. QuickDraw needs
; three pages.

        PushLong #0                      ;Space for handle
        PushLong #$300                  ;Three pages for QuickDraw
        PushWord MyID                   ;Owner
        PushWord #$C005                 ;Locked, fixed, fixed bank
        PushLong #0                     ;Location
        _NewHandle
        ldx #$FF
        jsr PrepareToDie
```

```

        pla                                ;Read handle and store in DP
        sta 0
        pla
        sta 2

        lda [0]                            ;Get dp location from handle

; Initialize QuickDraw

        pha                                ;Use dp obtained from handle
        PushWord #ScreenMode               ;Mode = 640
        PushWord #0                        ;Use entire screen
        PushWord MyID
        _QDStartup
        ldx #4
        jsr PrepareToDie

;-----
;
; Quickdraw's oval-drawing calls are RAM-based. Load them.

LoadAgain    PushWord #4                    ;QuickDraw is tool set 4
              PushWord #$0100
              _LoadOneTool
              bcc ToolsLoaded

              cmp #VolNotFound
              beq DoMount
              sec
              ldx #$FE
              jsr PrepareToDie

DoMount      anop
              jsr MountBootDisk
              cmp #1
              beq LoadAgain

              sec
              rts

; The tools have been loaded.

ToolsLoaded  clc                            ;Clear the carry flag
              rts                            ; and return

              END

;*****
;
; Define the bowtie region, then draw it.
;
;*****

DrawTie      START
              using GlobalData

; This is the bowtie region's definition

              PushLong #0                    ;Allocate space
              _NewRgn

```

230 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

```

pla                                ;Read the handle into TieHandle
sta TieHandle
pla
sta TieHandle+2

_OpenRgn                          ;Start defining the region

PushWord #60                      ;Move pen to starting point
PushWord #45                      ; (45,60)
_MoveTo

PushWord #30                      ;Draw the left-hand triangle
PushWord #60                      ; moving clockwise
_LineTo
PushWord #30
PushWord #30
_LineTo
PushWord #60
PushWord #45
_LineTo

PushLong #KnotDef                 ;Draw the knot
_FrameOval

PushWord #70                      ;Draw the right-hand triangle
PushWord #45                      ; moving clockwise
_MoveTo
PushWord #100
PushWord #30
_LineTo
PushWord #100
PushWord #60
_LineTo
PushWord #70
PushWord #45
_LineTo

PushLong TieHandle                ;End the region definition
_CloseRgn

; Set up the pen and draw the bowtie

PushWord #1                      ;Set the pen color to red
_SetSolidPenPat

PushLong TieHandle                ; and paint the bowtie
_PaintRgn

rts
END

;*****
;
; Event Loop
;
; Keep the bowtie on the screen until user presses a key.
;
;*****

```

```
EventLoop      START
```

```
WaitHere       lda $E0C000
                bpl WaitHere
                lda $E0C010
                rts
```

```
END
```

```
*****
*
* Prepare To Die
*
* Dies if carry is set.
*
* Error number to display is in X register.
* Assumes that Miscellaneous Tools is active.
*
*****
```

```
PrepareToDie    START
                bcs RealDeath    ;Carry = 1?
                rts              ; No. Return to caller

RealDeath       phx              ; Yes. Goodbye, program.
                PushLong #DeathMsg
                _SysFailMgr

DeathMsg        str 'Could not handle error '

END
```

```
;*****
;
; Mount Boot Disk
;
; Tells the user to insert the disk that contains the RAM-based
; tools.
;
;*****
```

```
MountBootDisk  START
```

```
    _Set_Prefix SetPrefixParams
    _Get_Prefix GetPrefixParams

    PushWord #0                ;Space for result
    PushWord #195              ;Column position for dialog box
    PushWord #30               ;Row position for dialog box
    PushLong #PromptStr        ;Prompt at top of dialog box
    PushLong #VolStr           ;Volume name string
    PushLong #OKStr            ;String in Button 1
    PushLong #CancelStr        ;String in Button 2
    _TLMountVolume

    pla                        ;Obtain the button number
    rts                        ; and return to caller
```

```

PromptStr      str 'Please insert the disk.'
VolStr         ds 16
OKStr          str 'OK'
CancelStr      str 'Shutdown'

GetPrefixParams dc i'7'
               dc i4'VolStr'
SetPrefixParams dc i'7'
               dc i4'BootStr'

BootStr        str '*/'

               END

```

Calculation Calls for Shapes

Earlier in this chapter, I described tool calls that move the pen to a specified location (MoveTo) or displace it a specified distance horizontally and vertically (Move). I also discussed LineTo and Line, which provide two ways to specify the end of a line. Similarly, QuickDraw also provides tool calls that move or displace shapes. These are handy for reproducing a shape at several places on the screen. To do this, you draw the shape, move it, then draw it again. The following are among the most useful calculation calls for shapes:

- SetRect and SetRectRgn change the corner coordinates of a rectangle or region to specified values, thereby moving it.
- OffsetRect, OffsetPoly, and OffsetRgn displace a rectangle, polygon, or region by specified horizontal and vertical values.
- CopyRgn copies the contents of one region into another.

A Color-Dithering Program

The DITHERS program in Example 7-4 employs OffsetRect to reproduce a rectangle called ColorBox at 16 positions across a 640-mode screen. DITHERS draws the leftmost box with a pen pattern of zeros (i.e., black), then enters a loop in which it displaces the box to the right, increments each pen pattern byte by 1, and redraws the box with the new pattern. The result is that each box has a different *dithered* color (see the earlier discussion of “Dithering in 640 Mode”). Box 0 is black, box 1 is blue, box 2 is gold, and so on.

Example 7-4

```

; DITHERS displays the 16 colors that can be obtained through dithering
; with the 640 mode's standard color table.

        absaddr on
        MCOPY Dithers.macros

Dithers      START
            using GlobalData

;-----
;
; Global equate used throughout the program.

ScreenMode   gequ $80                ;640 mode, no fill

            phk                      ;Set data bank to program
            plb                      ; bank to allow absolute addressing

            jsr InitStuff            ;Initialize everything
            bcs AllDone             ;Quit if initialization fails

            jsr ShowColors          ;Display the 16 color boxes

            jsr EventLoop           ;Wait for user to press a key

AllDone      anop                    ;All is done, shut down
            _QDS ShutDown           ;QuickDraw II
            _MT ShutDown            ;Miscellaneous Tools

            PushWord MyID           ;Discard the program's handle
            _DisposeAll

            PushWord MyID
            _MMShutdown             ;Memory Manager
            _TLShutdown            ;Tool Locator

            _Quit QuitParams        ;Do a ProDOS Quit call
            brk $F0                ;If it fails, break

        END

;*****
;
; Global Data
;
;*****

GlobalData   DATA

QuitParams   dc i4'0'              ;Return to caller
            dc i'$4000'            ;Make program restartable in memory

MyID         ds 2                  ;This will hold the program's i.d.

        END

```

```

;*****
;
; InitStuff
;
; Initializes tool sets and gets space in bank 0 for use as
; direct page by tools that need it.
;
;*****
InitStuff      START
              using GlobalData

; Initialize the Tool Locator, Memory Manager, and Miscellaneous
; Tools.

              _TLStartup

              PushWord #0
              _MMStartup

              pla                      ;Memory Manager returns program's ID
              sta MyID

              _MTStartup
              ldx #3
              jsr PrepareToDie

; Get some space for the direct page we need. QuickDraw needs
; three pages.

              PushLong #0              ;Space for handle
              PushLong #$300          ;Three pages for QuickDraw
              PushWord MyID           ;Owner
              PushWord #$C005         ;Locked, fixed, fixed bank
              PushLong #0             ;Location
              _NewHandle
              ldx #$FF
              jsr PrepareToDie

              pla                      ;Read handle and store in DP
              sta 0
              pla
              sta 2

              lda [0]                 ;Get dp location from handle

; Initialize QuickDraw

              pha                      ;Use dp obtained from handle
              PushWord #ScreenMode    ;Mode = 640
              PushWord #0             ;Use entire screen
              PushWord MyID
              _QDStartup
              ldx #4
              jsr PrepareToDie

              clc                      ;Clear the carry flag
              rts                      ; and return

END

```

```

;*****
;
; Show Colors
;
; Display the color boxes.
;
;*****

ShowColors START

        PushLong #PenPat           ;Set the pen pattern (i.e., color)
        _SetPenPat

        PushLong #ColorBox         ; and paint the leftmost box
        _PaintRect

; The following loop moves the color box 20 columns to the right,
; increments each pen pattern byte by one (to select the next
; dithered color), then paints the new color.

MoveBox  PushLong #ColorBox         ;Move the color box
        PushWord #20                ; 20 columns to the right
        PushWord #0
        _OffsetRect

BumpPat  ldx    #30                  ;Point to last 2 bytes in pattern
        clc                          ;Increment two bytes at a time
        lda    #$0101                ; by adding 1 to each of them
        adc    PenPat,x
        sta    PenPat,x
        dex
        dex                          ;Point to the preceding pair
        dex                          ; of bytes
        bpl    BumpPat

        PushLong #PenPat            ;Set the pen color
        _SetPenPat

        PushLong #ColorBox          ; and paint the next box
        _PaintRect

        dec    BoxCount              ;Any more boxes to paint?
        bne    MoveBox

        rts                          ;If not, exit

BoxCount dc i'15'

ColorBox dc i'5'                    ;Initial rectangle extends from
        dc i'5'                      ; (5,5) to (20,20)
        dc i'20'
        dc i'20'

PenPat   dc h'00 00 00 00 00 00 00 00' ;Pen pattern (initially black)
        dc h'00 00 00 00 00 00 00 00'
        dc h'00 00 00 00 00 00 00 00'
        dc h'00 00 00 00 00 00 00 00'

END

```

```

;*****
;
; Event Loop
;
; Keep the boxes on the screen until user presses a key.
;
;*****

EventLoop      START

WaitHere       lda $E0C000
               bpl WaitHere
               lda $E0C010
               rts

               END

*****
*
* Prepare To Die
*
* Dies if carry is set.
*
* Error number to display is in X register.
* Assumes that Miscellaneous Tools is active.
*
*****

PrepareToDie    START
               bcs RealDeath      ;Carry = 1?
               rts                ; No. Return to caller

RealDeath       phx                ; Yes. Goodbye, program.
               PushLong #DeathMsg
               _SysFailMgr

DeathMsg        str 'Could not handle error '

               END

```

By simply changing the ScreenMode constant from \$80 to 0, you can make DITHERS display the 16 colors in the 320 mode's standard color table. As mentioned earlier, that's the advantage of using an equate directive to specify the display mode. Change the equate and you change the mode throughout the program. (Changing ScreenMode also lets you switch color tables. There's more about color tables at the end of this chapter.)

Mouse Pointer Tool Call

QuickDraw also provides a *ShowCursor* tool that puts a very important shape on the screen: the mouse pointer. To call it, simply enter **__ShowCursor**.

The mouse comes into play in Chapter 9, where I discuss windows and the “controls” that operate on them.

Text

As you will see later, the Apple IIGS has tool sets that let you display attractive boxed prompts with on-screen buttons (perhaps *OK* and *Cancel*) that the user can “press” to respond. However, sometimes you may simply want to put a message on the screen without concern as to how it looks. For example, you may want to show “Please wait . . .” while the program performs a lengthy operation. In situations like this, you can use QuickDraw calls to display the text.

Table 7-10 lists the most useful tools for setting the foreground (text) and background colors and “drawing” the text. QuickDraw always draws text at the current pen location and uses the current foreground and background colors.

For example, to display “Please wait . . .”, your program would contain:

```

                PushLong #WaitStr
                _DrawString
                . .
                . .
WaitStr      str  'Please wait...'

```

Pixel Images

Up to now, you have seen how to draw lines and shapes using QuickDraw’s pen. However, sometimes you may want to display a picture that contains more than just the predefined shapes. For example, you may want to show a landscape with grass, trees, and flowers beneath a cloudy blue sky.

You could certainly draw such a picture 1 pixel at a time, using the pen. But, needless to say, that would be a difficult, tedious, and time-consuming task. You’d probably see drawing calls in your sleep!

How much easier it would be just to tell QuickDraw which color belongs at each location — to give it a pixel-by-pixel color “map” of your picture — then make it display the entire picture or some portion of it. Well, you can do just that by defining the picture as a *pixel image*. Pixel images

Table 7-10

Color Calls	
<code>__SetBackColor</code>	Set the background color
Call with: <code>PushWord ColorNum</code>	
<code>__SetBackColor</code>	
Result: None	
Note: QuickDraw only uses the appropriate number of bits in <code>ColorNum</code> . In 320 mode, it uses 4 bits, so <code>ColorNum</code> can range from 0 to 15. In 640 mode, it uses 2 bits, so <code>ColorNum</code> can be 0, 1, 2, or 3 (for black, red, green, or white, respectively).	
<code>__SetForeColor</code>	Set the foreground color
Call with: <code>PushWord ColorNum</code>	
<code>__SetForeColor</code>	
Result: None	
Note: See note for <code>SetBackColor</code> .	

Drawing Calls	
<code>__DrawChar</code>	Display the specified character
Call with: <code>PushWord #'char' ;Character</code>	
<code>__DrawChar</code>	
Result: None	
<code>__DrawString</code>	Display the specified Pascal-style string
Call with: <code>PushLong StringPtr ;Pointer to string</code>	
<code>__DrawString</code>	
Note: <code>StringPtr</code> points to a Pascal-style string. It has the form:	
<code>dc i1'count'</code>	
<code>dc 'string'</code>	
The “str” macro produces this format.	
<code>__DrawCString</code>	Display the specified C-style string
Call with: <code>PushLong StringPtr ;Pointer to string</code>	
<code>__DrawCString</code>	
Note: <code>StringPtr</code> points to a string of the form:	
<code>dc c'string'</code>	
<code>dc i1'0'</code>	

are easier to understand if you know something about their counterparts (called “bit maps”) in the Macintosh.

Macintosh Bit Maps

The term *bit map* reflects the fact that on a black-and-white Macintosh screen, pixels can only be “on” (black) or “off” (white) — and you can

represent those two states with a single bit. Thus, a bit map is a collection of 1's and 0's: a 1 for each black pixel and a 0 for each white one. Let's look at an example.

Figure 7-12 shows an enlargement of the "pencil" icon displayed by the Macintosh drawing program, *MacPaint*. Its image area consists of 288 pixel positions: 18 rows by 16 columns. Hence, the bit map for this image would be 288 bits long.

You could, of course, construct the bit map as a series of 18-bit patterns, each 16 bits long. However, entering sequences of 1's and 0's is both time-consuming and error-prone. Since each row is 16 columns wide, it's easier to construct the bit map using 16-bit word values.

As you know, a word contains 4 hexadecimal digits, where each digit represents 4 bit positions. To determine the pencil's bit map entry for a particular row, you must mentally divide the row into 4 quarters and enter the hex digit each quarter represents.

For example, the second row of the pencil image (shown in Figure 7-13) has four black boxes, which means it will contribute four 1's to the bit map entry.

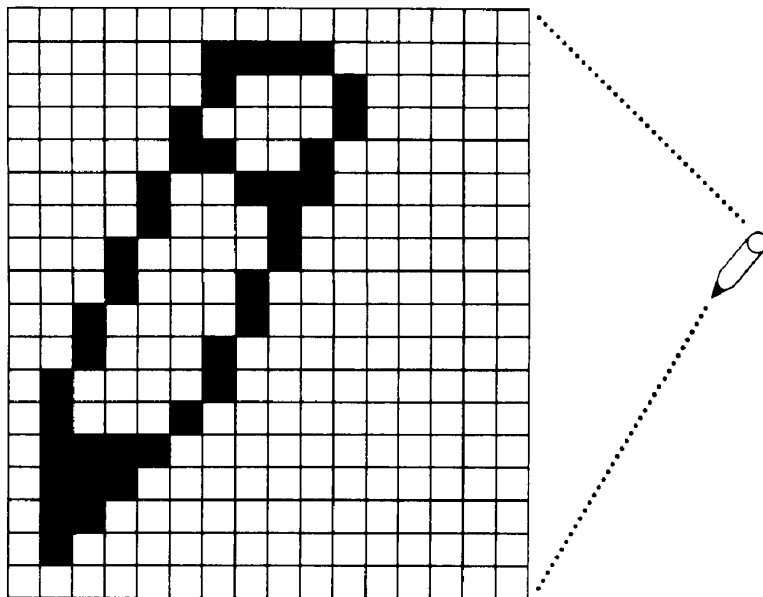


Figure 7-12

pictures on the Macintosh, because it only accommodates two colors: black (1) and white (0). But this approach is clearly inadequate for the Apple IIGS, because the IIGS must be able to display more than two colors at a pixel position. Specifically, it must display 16 colors in the 320 mode and four colors in the 640-mode. Obviously, then, a IIGS pixel image must be somewhat different from a Macintosh bit map.

Contents of a Pixel Image

What numeric values in a pixel image will produce colored pixels? As mentioned earlier, it takes a 4-bit pattern, or *nibble*, to select a color from the 16 entries in a 320-mode color table. As Table 7-2 shows, 0000 selects black, 0001 selects dark gray, and so on.

Let's return to my pencil image, with its 16-pixel rows. Bearing in mind that each pixel position must be represented by a 4-bit nibble (rather than a single bit), you need 16 nibbles — that is, 16 hex digits — to define a row. Assume that the pencil is to have a red body and black tip, and be displayed on a white background. In the 320-mode color table, red, black, and white have the pixel values \$7, \$0, and \$F, respectively. Thus, the first 8 pixels in the second row of the image are:

\$FFFF FF77

while the second eight pixels in this row are:

\$77FF FFFF

The hexadecimal values for the entire pencil image are shown in Figure 7-15.

The important point here is that *you must enter a color value for every pixel position in the image*. When displaying an image, QuickDraw does not apply the screen's current background pattern to unaffected pixels (as it does when drawing a boundary or frame with the pen), because there are no unaffected pixels!

Image Width

Since the pencil is only 10 pixels wide, you may wonder why the rows are 16 pixels wide. After all, the 5 columns of pixels to the right of it are extraneous; they only contain background data. The pencil image is 16 pixels wide because *QuickDraw can only work with images whose widths are multiples of 8 bytes*. Put another way, since there are 2 pixel numbers in

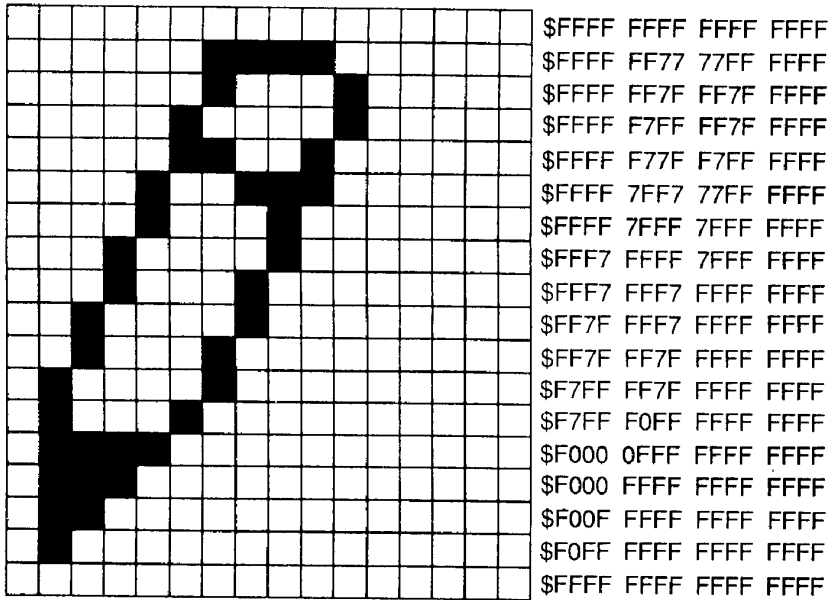


Figure 7-15

a byte, QuickDraw can only work with rows that are multiples of 16 pixels (that is, 16, 32, 48, and so on).

Fortunately, that doesn't mean that every pixel in the image is always displayed on the screen. You specify how much of the image is displayable by defining its "BoundsRect."

BoundsRect

The BoundsRect (short for boundary rectangle) is an imaginary frame that encloses the portion of an image that is to be available for display. For example, suppose you want to use only the first 12 columns of the pencil image — the pencil itself and the "background" column on its left and right. Further, suppose you want to situate the top left-hand corner of the displayable area at point (100,100) in the conceptual drawing space. To do this, you would assign the pencil image a BoundsRect that has the corner coordinates shown in Figure 7-16.

It's essential to realize that assigning a BoundsRect to a pixel image only positions it within the drawing space and scraps extraneous pixels. Assigning the BoundsRect does *not* display the image, nor does it even

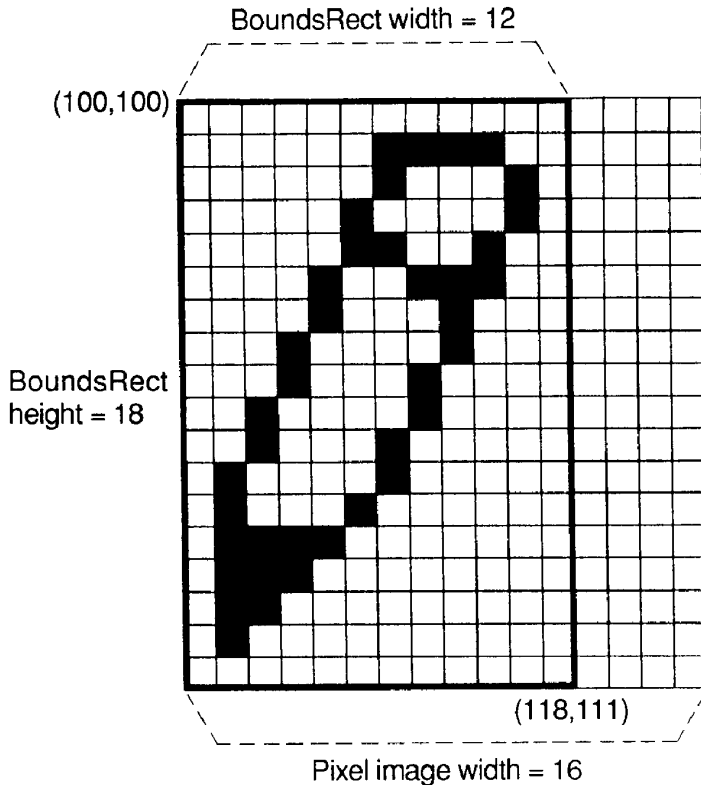


Figure 7-16

ensure that the image will ever be displayed. It simply makes the image *available* for display by QuickDraw.

To set up a BoundsRect, you must know something about the structure in which it dwells, the graphics port, or *GrafPort*.

The GrafPort

The GrafPort is a record of every parameter that relates to QuickDraw's current "personality" — how it's set to operate at the moment. The GrafPort keeps track of the current pen state, the foreground and background colors, the size and location of the BoundsRect, and so on. In other words, the GrafPort record oversees QuickDraw's *drawing environment*.

You cannot change an entry in the GrafPort record directly, but you do change one indirectly when you make a QuickDraw tool call that affects it. For example, SetPenSize and SetBackColor change the GrafPort's pen size and background color entries. In fact, whenever you make a QuickDraw tool call, QuickDraw checks the GrafPort to find out what pen size, background color, whatever, to use. QuickDraw tool calls are, in effect, calls to the GrafPort!

Entries in the GrafPort Record

Although it is not necessary to have expert knowledge of each and every detail of the GrafPort, you should have a general understanding of what it contains. Table 7-11 summarizes the entries in the GrafPort record.

Table 7-11. The GrafPort Record

PortInfo:	LocInfo
PortRect:	rect
ClipRgn:	handle
VisRgn:	handle
BkPat:	pattern
PnLoc:	point
PnSize:	point
PnMode:	integer
PnPat:	pattern
PnMask:	mask
PnVis:	integer
FontHandle:	handle
FontID:	longword
FontFlags:	integer
TxSize:	integer
TxFace:	style
TxMode:	integer
SpExtra:	fixed
ChExtra:	fixed
FGColor:	integer
BGColor:	integer
PicSave:	handle
RgnSave:	handle
PolySave:	handle
GrafProcs:	pointer
ArcRot:	integer
UserField:	long word
SysField:	long word

This first field, *PortInfo*, contains a data structure called *LocInfo* that is arranged as follows:

```
PortSCB: word
PointerToPixelImage: pointer
ImageWidth: word
BoundsRect: rect
```

Here, *PortSCB* is a word value that contains the *GrafPort*'s scan line control byte (SCB). The *PortSCB* does for the port what the master SCB does for QuickDraw overall (see Figure 7-5 for the SCB's format); that is, it specifies the graphics mode (320 or 640) and the color table.

The last three items in *LocInfo* incorporate your program's pixel data into the *BoundsRect*. *PointerToPixelImage* points to the data table for the pixel image, and *ImageWidth* specifies the width of the image's rows in bytes. Together, these two parameters tell QuickDraw where to find the image's data table and how to subdivide the long sequence of numbers it contains.

BoundsRect, the last *LocInfo* parameter, provides the corner coordinates of the image's boundary rectangle. These coordinates tell QuickDraw what portion of the conceptual drawing space to assign to the image. Further, the width of the *BoundsRect* tells QuickDraw how much of the image is available for display. (Remember, you can discard extraneous pixels by making the *BoundsRect* narrower than the pixel image table.)

The *PortRect*, the second item in the *GrafPort* record, is just as important as the *BoundsRect*. For an image or any part of it to actually be displayed — that is, to be visible on the screen — it must lie within *PortRect*'s border. Thus, the portion of an image that appears on the screen is the part that's inside both the *BoundsRect* and the *PortRect*. In other words, it is the area where these rectangles overlap or intersect.

To draw an analogy (and risk overstating my point), the *BoundsRect* is like a window in a house or office; it gives you the opportunity to see part of a much larger picture — the world outside. The *PortRect* is like a rectangular telescope through which you can look out the *BoundsRect* window.

How much you actually see through the telescope depends on its field of vision and whether it is pointing directly at the window. That is, what appears on the screen depends on the size of the *PortRect* and its orientation relative to the *BoundsRect*. If you still find this concept difficult to understand, Figure 7-17 should help make it clearer.

ClipRgn, short for clip region, lets you “lock” a portion of the *PortRect*

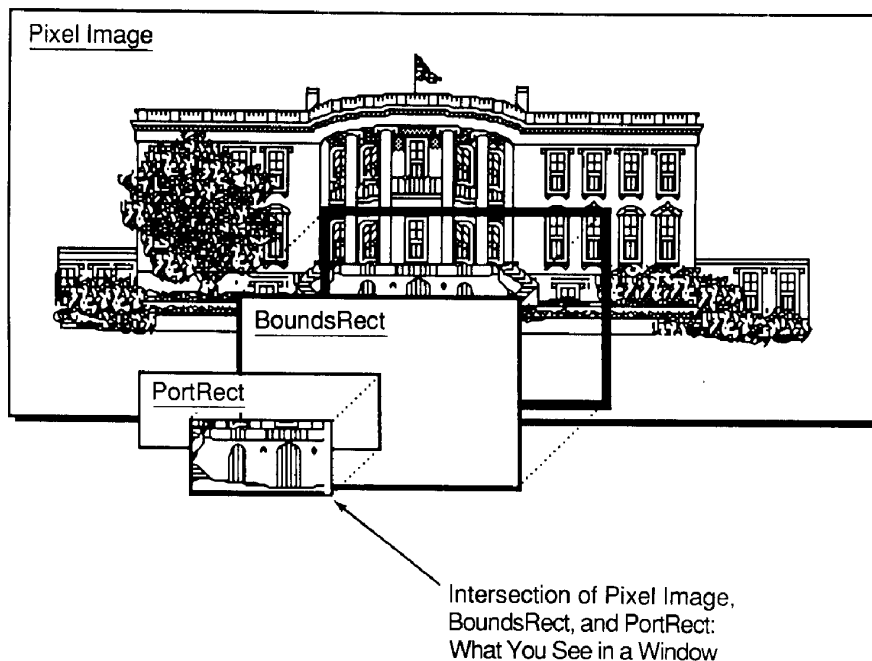


Figure 7-17

so that its pixels do not change. This can be handy in graphics programs that contain animation.

For example, imagine a game program that shows characters walking along a city street that has a large mailbox in the foreground. If the program has designated the mailbox area as a clip region, characters passing behind it would appear partially invisible. Without clipping, the program would have to change character pixels that are obscured by the mailbox; with a *ClipRgn* active, the obscured pixels are blocked out automatically. Although they are still within the intersection of the *BoundsRect* and *PortRect* — the normal conditions for being displayed — the fact that they are also within the clip region prevents them from appearing on the screen.

Note that *ClipRgn* is a region. It can be rectangular, like the *PortRect* and *BoundsRect*, but it need not be. As with regions you define for drawing with *QuickDraw*, the *ClipRgn* assumes whatever shape you give it.

VisRgn is used by the Memory Manager to display windows that overlap.

BkPat establishes the *GrafPort*'s background pattern. It can be changed with the calls *SetBackPat* and *SetSolidBackPat*.

You already understand *PnSize*, *PnMode*, *PnPat*, and *PnMask*. They are pen state attributes.

Sometimes you may want to deactivate the pen entirely, to disregard drawing calls in a certain part of the program. *PnVis* controls this condition. Call *HidePen* to deactivate the pen and *ShowPen* to reinstate it.

The next eight items, *FontHandle* through *ChExtra*, have to do with text and the font used to display it. *FGColor* and *BGColor* determine the foreground and background colors for text. They are affected by *SetForeColor* and *SetBackColor*.

You can generally ignore the remaining items in the record. They're used by system programs, not application programs.

Multiple GrafPorts

QuickDraw is not the only tool set that sets up a GrafPort; the *Window Manager* also sets one up whenever you open a new window. This makes it easy to work with multiple windows, because each has its own GrafPort — that is, its own unique drawing environment. Thus, when you switch between windows that overlap on the screen, the Window Manager activates the GrafPort for the window you want to work with. It thereby saves you the trouble of remembering the GrafPort settings for each individual window. Windows and the GrafPorts associated with them are discussed in Chapter 9.

Table 7-12 lists two tool calls you may need when working with multiple windows. The first, *GetPort*, returns a pointer to the current GrafPort. You use *GetPort* when you want to do something in another port (window), but need to remember this port so you can switch back to it later. *SetPort* is the call you use to switch ports.

Table 7-12

__GetPort		Get handle of the current GrafPort
Call with:	PushLong #0	;Space for result (handle)
	__GetPort	
Result:	Pointer to port (long word)	
__SetPort		Switch to the specified GrafPort
Call with:	PushLong <i>PortPtr</i>	;Pointer to port
	__SetPort	
Result:	None	

Displaying a Pixel Image

The preceding excursion into the world of the GrafPort gave me the opportunity to introduce the PortRect and describe its relationship to the BoundsRect. Let's take stock of where we are.

In the "Pixel Images" section, you learned how to set up a pixel image using a table of nibble-size, 320-mode color values. Since QuickDraw requires rows in images to be multiples of 8 bytes (or 16 pixels) wide, I introduced the BoundsRect, which lets you discard extraneous pixels. Because a BoundsRect only establishes a coordinate system for an image, however, it was necessary to introduce the PortRect. The intersection of the PortRect and BoundsRect determines what portion of the pixel image actually appears on the screen.

With all this theory presented, it's time to create a program that actually displays a pixel image. The pencil I have used all along is as good as any, so I'll stick with it.

Tool Calls to Display Pixel Images

To write a program that displays a pixel image, you must know which tool calls specify the image and set up the BoundsRect and PortRect. PPToPort, shown in Table 7-13, is the call you normally use to display an image.

The ScrollRect call moves a pixel image by specified horizontal and vertical displacements (dHoriz and dVert) within a rectangle to which RectPtr points. The displacements are signed numbers (integers), with positive values moving the image to the right or downward, respectively. The rectangle simply establishes the boundaries in which the image can move and still be visible. In other words, with ScrollRect, for an image to appear on the screen, it must lie within the intersection of the BoundsRect, PortRect, and the so-called scroll-area rectangle to which RectPtr points.

ScrollRect's calling sequence doesn't indicate *which* image it will move, or "scroll." That's because it obtains this information from the GrafPort automatically. That is, it always scrolls the image that is within the current BoundsRect and PortRect.

The final input to ScrollRect — UpRgnHandle — is simply the handle of a region in memory that QuickDraw will use as working storage for the move operation. ScrollRect does not create this region; you must provide it with a preceding NewRgn call. Since ScrollRect is primarily used to animate images, I will discuss it further in an upcoming "Animation" section.

Since PPToPort assumes you have already defined the PortRect (or accept the default, the entire screen), three PortRect calls are also listed.

Table 7-13

__PPToPort		Display a pixel image
Call with:	PushLong <i>LocInfoPtr</i> ;Pointer to parameter block PushLong <i>ImageRectPtr</i> ;Pointer to image rectangle PushWord <i>ScreenColumn</i> ;Screen column and PushWord <i>ScreenRow</i> ; row PushWord <i>PenMode</i> ;Pen transfer mode __PPToPort	
Result:	None	
Note:	LocInfoPtr points to: PortSCB (word) PointerToPixelImage (pointer) ImageWidth (word) BoundsRect (rect)	
__ScrollRect		Move a pixel image within a specified rectangle
Call with:	PushLong <i>RectPtr</i> ;Pointer to scroll area rect. PushWord <i>dHoriz</i> ;Horizontal displacement PushWord <i>dVert</i> ;Vertical displacement PushLong <i>UpRgnHandle</i> Handle of update region __ScrollRect	
Result:	None	
Note:	See text.	
__SetOrigin		Start PortRect at the specified point
Call with:	PushWord <i>Column</i> ;Column and PushWord <i>Row</i> ; row of top left corner __SetOrigin	
Result:	None	
__GetPortRect		Get the current PortRect coordinates
Call with:	PushLong <i>RectPtr</i> ;Pointer to buffer __GetPortRect	
Result:	None	
Note:	RectPtr points to: ds 2 ;Row and ds 2 ; column of top left corner ds 2 ;Row and ds 2 ; column of bottom right corner	
__SetPortRect		Set the PortRect coordinates
Call with:	PushLong <i>RectPtr</i> ;Pointer to PortRect coords. __SetPortRect	
Result:	None	
Note:	See note for GetPortRect.	

Note the similar calls `SetOrigin` and `SetPortRect`, which both change the PortRect. They differ in that `SetOrigin` only moves the upper left-hand corner of the PortRect (and leaves the bottom right-hand corner unchanged), whereas `SetPortrect` redefines it entirely.

Pencil Display Program

Example 7-5 lists a program called PENCIL that displays the pencil image (called PclImg here) on a 320 mode screen. In fact, PENCIL displays the pencil *twice*, at locations (0,0) and (50,50), on a light blue background. Here, the BoundsRect encloses only the first 12 columns (0-11) of the image, because columns 12 through 15 are extraneous.

Animation

In some applications you may want to move graphics objects (shapes or pixel images) on the screen, or *animate* them. Producing animation isn't as difficult as you might expect. In fact, it requires only five steps:

1. Display the object.
2. Wait a period of time, so the object is visible.
3. Erase the object.
4. Move to where you want the object to appear next.
5. Repeat the process, starting at Step 1.

Example 7-5

```
; PENCIL displays the pixel image of a pencil icon in 320 mode.

      absaddr on
      MCOPY Pencil.macros

Pencil      START
            using GlobalData

;-----
;
; Global equate used throughout the program.

ScreenMode      gequ 0                      ;320 mode, no fill

               phk                          ;Set data bank to program
               plb                          ; bank to allow absolute addressing

               jsr InitStuff                ;Initialize everything
               bcs AllDone                  ;Quit if initialization fails

               jsr ShowPencil               ;Display the pencil
```

```

                                jsr EventLoop                ;Wait for user to press a key

AllDone      anop                    ;All is done, shut down
              _QDShutDown            ;QuickDraw II
              _MTShutDown            ;Miscellaneous Tools

              PushWord MyID          ;Discard the program's handle
              _DisposeAll

              PushWord MyID
              _MMShutDown            ;Memory Manager
              _TLShutDown            ;Tool Locator

              _Quit QuitParams       ;Do a ProDOS Quit call
              brk $F0                ;If it fails, break

              END

;*****
;
; Global Data
;
;*****

GlobalData   DATA

QuitParams   dc i4'0'                ;Return to caller
              dc i'$4000'            ;Make program restartable in memory

MyID         ds 2                    ;This will hold the program's i.d.

              END

;*****
;
; InitStuff
;
; Initializes tool sets and gets space in bank 0 for use as
; direct page by tools that need it.
;
;*****

InitStuff    START
              using GlobalData

; Initialize the Tool Locator, Memory Manager, and Miscellaneous
; Tools.

              _TLStartup

              PushWord #0
              _MMStartup

              pla                    ;Memory Manager returns program's ID
              sta MyID

              _MTStartup
              ldx #3
              jsr PrepareToDie

```

252 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

```
; Get some space for the direct page we need. QuickDraw needs
; three pages.

    PushLong #0                ;Space for handle
    PushLong #$300            ;Three pages for QuickDraw
    PushWord MyID              ;Owner
    PushWord #$C005           ;Locked, fixed, fixed bank
    PushLong #0                ;Location
    _NewHandle
    ldx #$FF
    jsr PrepareToDie

    pla                        ;Read handle and store in dp
    sta 0
    pla
    sta 2

    lda [0]                    ;Get dp location from handle

; Initialize QuickDraw

    pha                        ;Use dp obtained from handle
    PushWord #ScreenMode       ;Mode = 640
    PushWord #0                ;Use entire screen
    PushWord MyID
    _QDStartup
    ldx #4
    jsr PrepareToDie

    clc                        ;Clear the carry flag
    rts                        ; and return

END

;*****
;
; Display the pencil at two places on the screen: (0,0) and (50,50).
;
;*****

ShowPencil START
    using PencilData

    PushWord #$BBBB            ;Set background to light blue
    _ClearScreen

    PushLong #PclImg            ;Store pencil image pointer
    pla
    sta Ptr2PI                  ; in LocInfo
    pla
    sta Ptr2PI+2

    PushLong #LocInfo           ;Pointer to parameter block
    PushLong #PencilRect        ;Pointer to source rectangle
    PushWord #0                  ;Start at column 0
    PushWord #0                  ; and row 0
    PushWord #0                  ;Use normal pen mode
    _PPToPort

    PushLong #LocInfo           ;Pointer to parameter block
    PushLong #PencilRect        ;Pointer to source rectangle
```

```

        PushWord #50                ;Start at column 50
        PushWord #50                ; and row 50
        PushWord #0                 ;Use normal pen mode
        _PToPort

        rts
    END

;*****
;
; Event Loop
;
; Keep the pencil on the screen until user presses a key.
;
;*****

EventLoop    START

WaitHere     lda $E0C000
             bpl WaitHere
             lda $E0C010
             rts

             END

;*****
;
; Pencil Data
;
; Pixel image and definition of the image.
;
;*****

PencilData DATA

PclImg      dc h'ffff ffff ffff ffff'      ;Row 1
             dc h'ffff ff77 77ff ffff'      ;Row 2
             dc h'ffff ff7f ff7f ffff'      ;Row 3
             dc h'ffff f7ff ff7f ffff'      ;Row 4
             dc h'ffff f77f f7ff ffff'      ;Row 5
             dc h'ffff 7ff7 77ff ffff'      ;Row 6
             dc h'ffff 7fff 7fff ffff'      ;Row 7
             dc h'ffff ffff 7fff ffff'      ;Row 8
             dc h'ffff7 fff7 ffff ffff'      ;Row 9
             dc h'fff7f fff7 ffff ffff'      ;Row 10
             dc h'fff7f fff7 ffff ffff'      ;Row 11
             dc h'f7ff fff7 ffff ffff'      ;Row 12
             dc h'f7ff f0ff ffff ffff'      ;Row 13
             dc h'f000 0fff ffff ffff'      ;Row 14
             dc h'f000 ffff ffff ffff'      ;Row 15
             dc h'f00f ffff ffff ffff'      ;Row 16
             dc h'f0ff ffff ffff ffff'      ;Row 17
             dc h'ffff ffff ffff ffff'      ;Row 18

PencilRect  dc i'0,0,17,15'      ;Coordinates of pencil image

; This is the LocInfo to be sent to the GrafPort

```

```

LocInfo  anop
PortSCB  dc i'0'                                ;SCB
Ptr2PI   ds 4                                    ;Pointer to image
Width    dc i'8'                                ;Width of image (bytes)
BoundsRect dc i'0,0,17,11'                      ;BoundsRect

        END

*****
*
* Prepare To Die
*
* Dies if carry is set.
*
* Error number to display is in X register.
* Assumes that Miscellaneous Tools is active.
*
*****

PrepareToDie  START
               bcs RealDeath      ;Carry = 1?
               rts                ; No. Return to caller

RealDeath     phx                  ; Yes. Goodbye, program.
               PushLong #DeathMsg
               _SysFailMgr

DeathMsg      str 'Could not handle error '

        END

```

Generating a wait interval or delay, as prescribed in Step 2, takes some non-QuickDraw tool calls that I'll describe shortly. The instructions and tool calls you need for the other steps depend on what kind of objects you are animating — shapes or pixel images.

Animating Shapes

You can probably guess which calls you need to animate a QuickDraw shape. You need a “Frame,” “Paint,” or “Fill” type call to draw it, an “Erase” type call to erase it, and a “Set” or “Offset” type call to move it to the next display position. For example, you need `EraseRgn` to erase a region and either `SetRectRgn` or `OffsetRgn` to move it.

Animating Pixel Images

Believe it or not, pixel images are somewhat easier to animate than regular shapes, because you can use a single tool call to erase the image and move

it to its next position. The tool that does these jobs is *ScrollRect* (summarized in Table 7-13). *ScrollRect* not only erases the image and moves it, but it also updates the image's location.

Example 7-6 lists a program called ANIMATE that moves the pencil icon across a 320 mode screen. ANIMATE is simply a modified version of PENCIL (Example 7-5) in which the *MoveImg* segment (called *ShowPencil* before) contains a *NewRgn* call to allocate a scroll area in memory for the image and a *ScrollRect* call to do the scrolling.

Example 7-6

```
; ANIMATE moves a pencil icon across a 320 mode screen.

      absaddr on
      MCOPY Animate.macros

Animate      START
             using GlobalData

;-----
;
; Global equate used throughout the program.

ScreenMode   gequ 0                      ;320 mode, no fill

             phk                          ;Set data bank to program
             plb                          ; bank to allow absolute addressing

             jsr InitStuff                ;Initialize everything
             bcs AllDone                 ;Quit if initialization fails

             jsr MoveImg                  ;Do the animation

             jsr EventLoop                ;Wait for user to press a key

AllDone      anop                        ;All is done, shut down
             _QDShutdown                  ;QuickDraw II
             _MTShutdown                  ;Miscellaneous Tools

             PushWord MyID                 ;Discard the program's handles
             _DisposeAll

             PushWord MyID
             _MMShutdown                  ;Memory Manager
             _TLShutdown                  ;Tool Locator

             _Quit QuitParams             ;Do a ProDOS Quit call
             brk $F0                      ;If it fails, break

END
```

256 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

```
;*****
;
; Global Data
;
;*****

GlobalData    DATA

ImgHandle     ds 4                      ;This will hold image's handle

QuitParams    dc i4'0'                  ;Return to caller
               dc i'$4000'              ;Make program restartable in memory

MyID          ds 2                      ;This will hold the program's i.d.

               END

;*****
;
; InitStuff
;
; Initializes tool sets and gets space in bank 0 for use as
; direct page by tools that need it.
;
;*****

InitStuff     START
               using GlobalData

; Initialize the Tool Locator, Memory Manager, and Miscellaneous
; Tools.

               _TLStartup

               PushWord #0
               _MMStartup

               pla                      ;Memory Manager returns program's ID
               sta MyID

               _MTStartup
               ldx #3
               jsr PrepareToDie

; Get some space for the direct page we need. QuickDraw needs
; three pages.

               PushLong #0              ;Space for handle
               PushLong #$300           ;Three pages for QuickDraw
               PushWord MyID            ;Owner
               PushWord #$C005          ;Locked, fixed, fixed bank
               PushLong #0              ;Location
               _NewHandle
               ldx #$FF
               jsr PrepareToDie

               pla                      ;Read handle and store in dp
               sta 0
               pla
               sta 2
```



```

        lda [0]                ;Get dp location from handle
; Initialize QuickDraw

        pha                    ;Use dp obtained from handle
        PushWord #ScreenMode   ;Mode = 640
        PushWord #0            ;Use entire screen
        PushWord MyID
        _QDStartup
        ldx #4
        jsr PrepareToDie

        clc                    ;Clear the carry flag
        rts                    ; and return

        END

;*****
;
; Move Image
;
; Display the pencil, then move it.
;
;*****

MoveImg START
using GlobalData
using PencilData

        PushWord #$BBBB        ;Set background to light blue
        _ClearScreen

        PushWord #$B           ; and BackPat to same color
        _SetSolidBackPat

        PushLong #PclImg       ;Store pencil image pointer
        pla
        sta Ptr2PI             ; in LocInfo
        pla
        sta Ptr2PI+2

; Display the pencil at its starting position (0,0).

        PushLong #LocInfo      ;Pointer to parameter block
        PushLong #PencilRect   ;Pointer to source rectangle
        PushWord #0            ;Start at column 0
        PushWord #0            ; and row 0
        PushWord #0            ;Use normal pen mode
        _PPToPort

        PushLong #0            ;Set up a region for the image
        _NewRgn
        pla                    ;Read the handle into ImgHandle
        sta ImgHandle
        pla
        sta ImgHandle+2

; Move the image to the right 60 times, advancing 5 point positions
; each time.

```

258 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

```

MoveAgain anop
    PushLong #ScrollRect        ;Pointer to scroll rectangle
    PushWord #5                 ;Move the image 5 columns
    PushWord #0                 ; to the right
    PushLong ImgHandle          ;Specify its region's handle
    _ScrollRect

    dec MoveCount
    bne MoveAgain

    rts

MoveCount dc i'60'              ;Move count variable

END

;*****
;
; Event Loop
;
; Keep the pencil on the screen until user presses a key.
;
;*****

EventLoop    START

WaitHere     lda $E0C000
             bpl WaitHere
             lda $E0C010
             rts

             END

;*****
;
; Pencil Data
;
; Pixel image and definition of the image.
;
;*****

PencilData DATA

PclImg       dc h'ffff ffff ffff ffff'        ;Row 1
             dc h'ffff ff77 77ff ffff'        ;Row 2
             dc h'ffff ff7f ff7f ffff'        ;Row 3
             dc h'ffff f7ff ff7f ffff'        ;Row 4
             dc h'ffff f77f f7ff ffff'        ;Row 5
             dc h'ffff 7fff 77ff ffff'        ;Row 6
             dc h'ffff 7fff 7fff ffff'        ;Row 7
             dc h'fff7 ffff 7fff ffff'        ;Row 8
             dc h'fff7 ffff ffff ffff'        ;Row 9
             dc h'fff7 ffff ffff ffff'        ;Row 10
             dc h'fff7 ff7f ffff ffff'        ;Row 11
             dc h'f7ff ff7f ffff ffff'        ;Row 12
             dc h'f7ff f0ff ffff ffff'        ;Row 13
             dc h'f000 0fff ffff ffff'        ;Row 14
             dc h'f000 ffff ffff ffff'        ;Row 15

```

```

        dc h'f00f ffff ffff ffff'      ;Row 16
        dc h'f0ff ffff ffff ffff'      ;Row 17
        dc h'ffff ffff ffff ffff'      ;Row 18

PencilRect dc i'0,0,17,15'      ;Coordinates of pencil image
ScrollRect dc i'0,0,17,320'     ;Scroll rectangle

; This is the LocInfo to be sent to the GrafPort

LocInfo anop
PortSCB dc i'0'                  ;SCB
Ptr2PI ds 4                      ;Pointer to image
Width dc i'8'                    ;Width of image (bytes)
BoundsRect dc i'0,0,17,11'      ;BoundsRect

END

*****
*
* Prepare To Die
*
* Dies if carry is set.
*
* Error number to display is in X register.
* Assumes that Miscellaneous Tools is active.
*
*****

PrepareToDie START
        bcs RealDeath      ;Carry = 1?
        rts                ; No. Return to caller

RealDeath phx                ; Yes. Goodbye, program.
          PushLong #DeathMsg
          _SysFailMgr

DeathMsg str 'Could not handle error '

END

```

Note that the PushWord #5 input to ScrollRect makes it move the pencil five columns to the right each time. I arrived at this number through some trial and error. At the outset, I tried moving the image one column at a time. The image moved fairly smoothly, so the animation looked fine, but it was tortuously slow. The pencil took about 35 *seconds* to cross the screen!

Next I tried moving the image an entire width (18 points) at a time. It then took about 3 seconds to cross the screen, but the movement was very jumpy. After more experimenting, I finally settled on moving five points at a time. That seems close to optimal for both speed and appearance.

Time and Date Operations

The delay interval needed in an animation routine to make a displayed object visible is just one of many operations that pertain to the time or date. For instance, you may want to display the elapsed time or the remaining time for a game that involves specific intervals. Obvious examples are football and basketball simulations; each game uses a clock to show the time remaining in a quarter or period. Similarly, a computerized chess game may include a timer that requires the player to move within some set period. Educational programs often include timers for the same purpose — to set a limit on the amount of time a student can use to solve a set of problems.

These examples all involve intervals of time. You may also want to use the current time and date directly, to “time stamp” a document.

These are just a few situations where having access to the time and date is handy, and you can probably think of many others. At any rate, be thankful for the battery backed-up clock that’s built into the Apple IIGS; it’s the key to performing just about any time- or date-related job you want to do.

Tool Calls for Reading the Time and Date

The Apple IIGS Miscellaneous Tools set contains tools that return the current time and date, as stored in the battery backed-up RAM. Built into ROM, Miscellaneous Tools contains what you might expect: tools that don’t fall into any category that’s broad enough to warrant its own tool set. Included are tools that interface with the mouse, read and change parameters in the battery backed-up RAM (the parameters used by the Control Panel), and do a variety of other jobs.

However, the only tools that are of interest here are *ReadTimeHex* and *ReadAsciiTime*, which return the time and date on the stack or as a text string, respectively. Table 7-14 summarizes these calls and the start-up and shutdown calls for the Miscellaneous Tools.

ReadTimeHex returns 4 words, with a different parameter in each byte. Here is a description of the words, listed in the order you pull them off the stack:

- The first word contains the Minute (high byte) and Second values.
- The second word contains the Year (minus 1900) and Hour (0-23) values.
- The third word contains the Month and Day values. The Month can range from 0 to 11, where 0 is January.

Table 7-14

<u>MTStartup</u>		Start the Miscellaneous Tools
Call with: <u>MTStartup</u>		
Result: None		
<u>MTShutDown</u>		Shut down the Miscellaneous Tools
Call with: <u>MTShutDown</u>		
Result: None		
<u>ReadTimeHex</u>		Read the time and date in hex format
Call with: PushWord #0		;Space for 4 result words
PushWord #0		
PushWord #0		
PushWord #0		
<u>ReadTimeHex</u>		
Results: Four word values; see text.		
<u>ReadAsciiTime</u>		Read the time and date in ASCII format
Call with: PushLong <i>StringPtr</i>		;Pointer to string buffer
<u>ReadAsciiTime</u>		
Result: None		
Note: Returns 20 characters, with the high-order bit of each character set to one.		
See text for string formats.		

- The high byte of the fourth word contains the Day of the Week (0-6, where 0 is Sunday); its low byte is unused and contains 0.

Note that Read Time Hex can only produce the time to the nearest second; it doesn't report hundredths or even tenths of a second, so it wouldn't be very useful as a stopwatch. To get that degree of accuracy, you would have to use another Miscellaneous Tools call, *GetTick*. The so-called tick count that *GetTick* returns gets incremented 60 times a second.

The other date-and-time call, *ReadAsciiTime*, returns the date and time as a 20-character ASCII text string. The format of this string depends on which date and time format is active in the Control Panel's "Clock" options. The possible formats are as follows (with the first one being the default):

Date Format	Time Format	ReadAsciiTime String		
MM/DD/YY	AM-PM	mm/dd/yy	HH:MM:SS	AM or PM
DD/MM/YY	AM-PM	dd/mm/yy	HH:MM:SS	AM or PM
YY/MM/DD	AM-PM	yy/mm/dd	HH:MM:SS	AM or PM
MM/DD/YY	24 Hour	mm/dd/yy	HH:MM:SS	
DD/MM/YY	24 Hour	dd/mm/yy	HH:MM:SS	
YY/MM/DD	24 Hour	yy/mm/dd	HH:MM:SS	

Here, the “24 Hour” time format indicates international time, with hours ranging from 0 (midnight) to 23 (11:00 P.M.).

Standard ASCII characters are 8-bit values in which the high-order bit is set to 0 (see Appendix B). However, ReadAsciiTime’s characters have the high bit set to 1! This means that if you want to display the string, you have to strip off the high bit from each character byte. Example 7-7 shows a sub-routine called GetTime that does this.

Earlier, I promised to talk about generating delays, so here it is.

Generating Delays

Programs that generate delays usually do the following:

1. Read the current time.
2. Add selected increments to the time, to produce a “target” time — the time at which the delay should end.

Example 7-7

```
GetTime  START

          PushLong #TimeString      ;Read the time and date
          _ReadAsciiTime

; ReadAsciiTime returns characters with the high bit set to 1.
; I must make these bits 0 to display the string.

NextWord  ldx #18                    ;Strip off the high bit
          lda TimeString,x          ; 2 bytes at a time
          and #$7F7F
          sta TimeString,x
          dex
          dex
          bpl NextWord

          PushWord #10               ;Move the pen to (10,10)
          PushWord #10
          _MoveTo

          PushLong #TimeString      ; and display the string
          _DrawCString

          rts

TimeString ds 20
          dc il'0'

END
```

3. Adjust the target time so that hours don't exceed 23 and minutes and seconds don't exceed 59. This involves carrying any excess to the next higher unit.
4. Read the time repeatedly until the current time matches or exceeds the target time.

Figure 7-17 shows a flowchart for producing a time delay based on user-specified increments which are added to the minute and second values. (I assume you won't want to delay for hours at a time, although you could.)

So now the job at hand is to convert the flowchart into a usable program. In essence, the program should consist of a couple of `ReadTimeHex` calls separating assembly language instructions that add the increments to the initial time and then adjust the target time so that it contains legal values. The subroutine in Example 7-8, called `DELAY`, shows one way of doing these jobs. `DELAY` obtains the duration of the delay from `A` (seconds) and `Y` (minutes).

Although the listing is longer than one might expect for such a simple job, most of the bulk results from the fact that `ReadTimeHex` packs its time and date values into individual bytes of a word. Since byte data is awkward to work with in 16-bit native mode, the results were unpacked and stored in word variables. Then, after the target time was calculated, the words were repacked into `ReadTimeHex`'s byte-oriented format.

Because `DELAY` is based on `ReadTimeHex`, the delays it produces may not be very accurate. For example, if you request a 3-second delay when the current time is 8:29:00:90 (that is, 90/100 of a second past 8:29), `ReadTimeHex` ignores the fraction and returns 8:29. In response, `DELAY` adds 3 seconds and sets the target time to 8:32. However, 1/10 of a second later, the time changes to 8:30, which means you actually receive a delay of 2.10 seconds — far short of what you want.

The moral of the story is: *DELAY's accuracy may be off by nearly a second.* If this is unacceptable, you may want to resort to `Get Tick` is accurate to 1/60 of a second.

To use `DELAY` in a program, simply call it with a sequence of the form:

```
lda seconds      ;Number of seconds to wait
ldy minutes      ;Number of minutes to wait
jsr Delay        ;Start waiting
```

and put a *copy delay.src* statement at the end of your program.

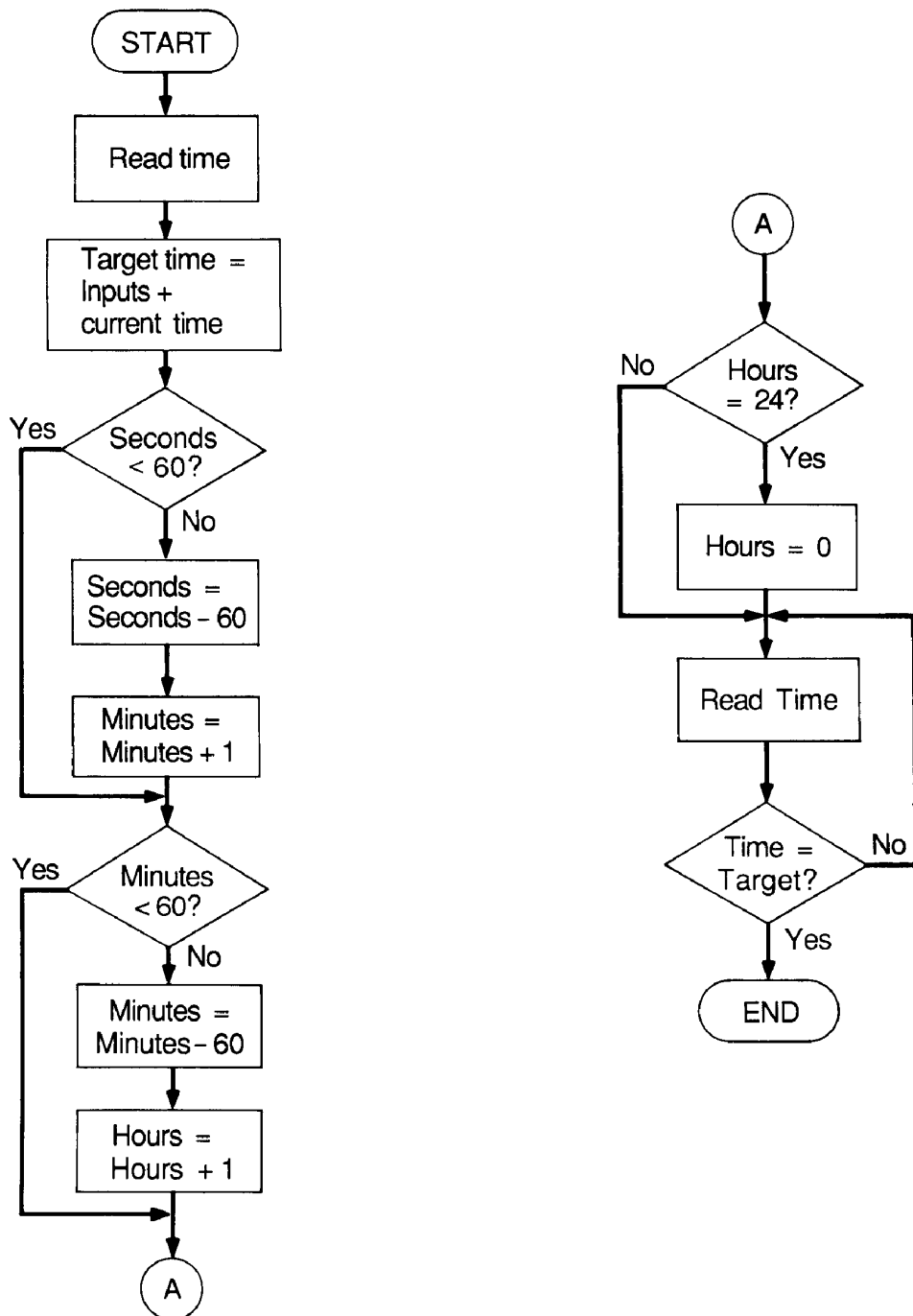


Figure 7-17

Example 7-8

```

;*****
;
; DELAY generates a delay interval based on a seconds count in the
; accumulator and a minutes count in the Y register.
; The subroutine uses the Miscellaneous Tools set, so the calling
; program must include _MTStartup and _MTShutDown.
;
;*****

        mcopy delay.macros

Delay  START

        phy                                ;Preserve the inputs during
        pha                                ; the ReadTimeHex call

        PushWord #0                        ;Read the time and date
        PushWord #0
        PushWord #0
        PushWord #0
        _ReadTimeHex

; Time and date values are on the stack. Remove them and store the
; second, minute, hour, and year values in memory. Discard the month,
; day, and day of the week.

        pla                                ;Get minutes and seconds
        tay                                ;Save this value temporarily
        and #$FF                           ; while I strip off the minutes
        sta OrigSecs                       ; and save the seconds
        tya                                ;Now retrieve the minutes value,
        xba                                ; put it in the low byte,
        and #$FF                           ; strip off the seconds,
        sta OrigMins                       ; and save the minutes

        pla                                ;Get years and hours
        tay                                ;Save this value temporarily
        and #$FF                           ; while I strip off the years
        sta OrigHrs                       ; and save the hours.
        tya                                ;Now retrieve the years value,
        xba                                ; put it in the low byte,
        and #$FF                           ; strip off the hours,
        sta OrigYrs                       ; and save the years

        pla                                ;Discard month and day,
        pla                                ; and day of the week

; Calculate the seconds value for the target time.

        pla                                ;Retrieve the seconds input
        clc                                ; and add the current secs. to it
        adc OrigSecs
        cmp #60                           ;Seconds < 60?
        blt SaveSecs
        sec                                ; No. Subtract 60
        sbc #60
        inc OrigMins                       ; and increment the minutes value
SaveSecs sta OrigSecs                     ;Save the final seconds value

```

266 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

; Calculate the minutes value for the target time.

```

        pla                ;Retrieve the minutes input
        clc                ; and add the current mins. to it
        adc OrigMins
        cmp #60            ;Minutes < 60?
        blt SaveMins
        sec                ; No. Subtract 60
        sbc #60
        inc OrigHrs        ; and increment the hours value
SaveMins sta OrigMins      ;Save the final minutes value

```

; Adjust the hours value, if necessary.

```

        lda OrigHrs
        cmp #24            ;Hours = 24?
        bne Repack
        stz OrigHrs        ; If so, set hours to zero

```

; Pack the final values into the byte-oriented format that
; ReadTimeHex produces.

```

Repack  lda OrigMins        ;Minutes and seconds
        xba
        ora OrigSecs
        sta TargetMinSec
        lda OrigYrs        ;Years and hours
        xba
        ora OrigHrs
        sta TargetYrHour

```

; Read the time until it is the same as or greater than the target.

```

NewTime PushWord #0        ;Read the time and date again.
        PushWord #0
        PushWord #0
        PushWord #0
        _ReadTimeHex

```

```

        pla                ;Put min/sec word in A
        ply                ; and year/hr word in Y
        plx                ;Ignore the other two words
        plx

```

```

        cpy TargetYrHour   ;Do year/hr words match?
        beq TryMS
        bcs ThatsAll
TryMS   cmp TargetMinSec    ; If so, compare min/sec words
        blt NewTime

```

```

ThatsAll rts                ;Exit

```

```

OrigSecs ds 2
OrigMins ds 2
OrigHrs  ds 2
OrigYrs  ds 2

```

```

TargetMinSec ds 2
TargetYrHour ds 2

```

END

Note that you need not assemble `DELAY.SRC` separately; the `Copy` statement makes the assembler append it to your program, then assemble it along with your own statements. However, you must still generate `DELAY`'s macro library (`DELAY.MACROS`) using `MACGEN`; an *mcopy* statement at the beginning of the subroutine reads the library when you assemble.

`DELAY` is just one example of code you might consider keeping separate, so you can access it from any program. In fact, I recommend putting *any* job you do frequently into a file of its own. That way, you build a program library of useful routines, and you needn't "reinvent the wheel" with each new program. Having working, debugged code in separate modules also makes your program listings shorter and (unless you have used *lots* of external code) easier to read.

Working with Color Tables

Within the Apple IIGS memory are 32 color tables: 16 tables for the 640 mode and another 16 for the 320 mode. So far, I have been working with table 0, the default, or "standard," color table for each mode, but you can easily make QuickDraw use any of the other 15.

Recall from Figure 7-5 that the low 4 bits of the *scan line control byte* (SCB) select the color table. Therefore, to start QuickDraw in 640 mode with, say, color table 5, you would push an SCB value of \$85 — rather than \$80 — onto the stack before calling `QDStartup`.

You can also switch color tables within a program by calling `SetMasterSCB` (see Table 7-15). For example, to switch to table 5 in 640 mode with no fill, enter:

```
PushWord #$85      ;640 mode, no fill, table 5
_SetMasterSCB
```

As I write this, only the first seven predefined color tables (tables 0 through 6) for each mode actually hold useful color data; the rest (tables 7 through 15) are filled with zeros. Tables 7-2 and 7-3 summarized the standard color table (0) for each mode; Table 7-16 summarizes the color data for tables 1 through 6 in 640 mode. (Only minipalettes 0 and 1 are shown; minipalettes 2 and 3 are identical to 0 and 1.) Note that in each case, only the Pixel 1 entries differ from the standard color table.

To actually *see* the dithered colors one of these alternate tables can produce, run my `DITHERS` program (Example 7-4) with the appropriate SCB

Table 7-15

__SetColorTable		Create a new color table
Call with:	PushWord <i>TableNumber</i> ;Number for new table (1-15) PushLong <i>TablePtr</i> ;Pointer to table __SetColorTable	
Result:	None	
Note:	TablePtr points to a 16-word table whose entries specify the color values for the pixel values 0 through 16.	
__InitColorTable		Get a copy of the standard color table
Call with:	PushLong <i>TablePtr</i> ;Pointer to table buffer __InitColorTable	
Result:	None	
Note:	TablePtr points to: ds 32 ;16-word buffer for table	
__GetColorTable		Copy one color table into another
Call with:	PushWord <i>TableNumber</i> ;Table to be copied (0-15) PushLong <i>DestTblPtr</i> ;Pointer to dest. table __GetColorTable	
Result:	None	
Note:	DestTblPtr points to: ds 32 ;16-word buffer for table	
__SetColorEntry		Set the value of a color in a color table
Call with:	PushWord <i>TableNumber</i> ;Table number (0-15) PushWord <i>EntryNumber</i> ;Pixel value (0-15) PushWord <i>NewColor</i> ;Master color value __SetColorEntry	
Result:	None	
__SetMasterSCB		Set the master scan line control byte
Call with:	PushWord <i>SCBValue</i> ;SCB value __SetMasterSCB	
Result:	None	
Note:	See the earlier Figure 7-5 for the layout of the SCB and note that bits 0 through 3 specify the color table.	

value in the ScreenMode equate. For example, to obtain the colors for table 5, set ScreenMode to \$85.

Color tables 1 through 6 for 320 mode contain the *same* color values as they do for 640 mode! (Again, this will probably change with time.) That is, the first eight entries in the 320 mode's table 1 produce black, light green, green, white, black, light green, yellow, and white, respectively — and so do the last eight entries in that table.

If none of the predefined color tables suit your needs, you can create your own. Since color values are 3 hex digits, or 12 bits long, they offer

Table 7-16

Pixel Value	Color	Master Color Value	Color	Master Color Value
<i>Table 1</i>		<i>Table 2</i>		
0	Black	000	Black	000
1	Light Green	0C0	Yellow	FF0
2	Green	0F0	Green	0F0
3	White	FFF	White	FFF
0	Black	000	Black	000
1	Light Green	0C0	Yellow	FF0
2	Yellow	FF0	Yellow	FF0
3	White	FFF	White	FFF
<i>Table 3</i>		<i>Table 4</i>		
0	Black	000	Black	000
1	Orange	F80	Red	F00
2	Green	0F0	Green	0F0
3	White	FFF	White	FFF
0	Black	000	Black	000
1	Orange	F80	Red	F00
2	Yellow	FF0	Yellow	FF0
3	White	FFF	White	FFF
<i>Table 5</i>		<i>Table 6</i>		
0	Black	000	Black	000
1	Rose	F0F	Bluish Black	004
2	Green	0F0	Green	0F0
3	White	FFF	White	FFF
0	Black	000	Black	000
1	Rose	F0F	Bluish Black	004
2	Yellow	FF0	Yellow	FF0
4	White	FFF	White	FFF

4,096 different number combinations. Hence, that's how many colors are available — 4,096.

You can create a new color table from scratch by issuing a `SetColorTable` call. `SetColorTable` requires you to supply the number of the new table and a pointer to a 16-word data structure containing the color values. Give your table any number from 1 to 15; don't number it 0, because that refers to the standard color table. Finally, give a `SetMasterSCB` to make QuickDraw use your new table.

If the standard color table is close to what you want, copy it into memory with an `InitColorTable` call, then assign the copy a table number with `SetColorTable`. Finally, use `SetColorEntry` calls to change entries in the copy to what you want, then `SetMasterSCB` to activate the table.

You can also use this procedure to create a new color table from an existing one that's similar. Simply give a `GetColorTable` call instead of `InitColorTable`.

CHAPTER 8

Events

In Apple IIGS terminology, an *event* is anything that requires the computer's attention. This is normally something the user does, such as pressing a key or the mouse button, but it could also be, say, a character coming in through the serial port. (No, it *doesn't* include a spider coming in through an expansion slot cutout on the back panel!)

The Event Manager keeps track of every event that occurs, and records information about it in an *event record* in memory. In most cases, the Event Manager also logs the type of event that occurred (e.g., key press or mouse button press) in an *event queue* in memory. The application program's job is, then, to check the event queue periodically and respond to any events that are meaningful to the program. This unique way in which the IIGS handles key presses, mouse clicks, and other events means that the structure of your programs must be somewhat different from programs you have written in the past.

Modal Programs

Most people write programs that are very mode-oriented. Consider, for example, a typical computer-assisted education program that drills a student in mathematics. The program starts by displaying a menu with the choices

Addition, Subtraction, Multiplication, Division, and Quit. When the student has selected an option, the program presents perhaps ten problems, one at a time, and the student enters the answer to each one. The student is then asked whether he or she wants to receive another ten problems or return to the menu.

Such a program is very inflexible. Once it starts presenting problems, the student has only two options: solve all ten of them or turn the computer off. And when the “Do you want more problems? Y/N” prompt appears, the student is forced to make another all-or-nothing decision — ten more problems or quit. Maybe he or she would like to continue, but wants only two or three problems. Unfortunately, the program doesn’t provide that choice.

Just as the student is always locked into one of several modes (choose from the menu, answer problems, or respond to a prompt), so is the program. It is entirely dedicated to displaying a menu, problem, or prompt, and processing the user’s response. While waiting for a response, the program does nothing; it just sits idle, locked in a mode.

Modal programs are probably so distracting because we humans don’t operate in modes; we can do several things at a time. For example, some people jot notes while they talk on the telephone or read a magazine while they soak in the bathtub. My teenagers even claim to be able to do their homework while watching TV! People don’t operate in modes, and neither should your programs.

In every program you write, then, you should allow the user to select any program function at any time. That is, the user should be able to pull down a menu, move things around on the screen, or do anything else when he or she *wants to*, without putting the program in a special user-interface mode. There should be no such modes. Instead, the program should regularly check, or *poll*, the event queue to find out whether the user has asked for the program’s attention.

For example, if the mouse pointer has been clicked in a box that closes a window, the Event Manager will put an entry in its event queue to reflect that action. In this case, upon polling the event queue and finding the mouse event, the program should remove the window from the screen immediately. Similarly, if the user has pressed a key in a program that displays text, the Event Manager will put a code for the key in the event queue. Upon finding it, your program should send that character to the screen.

In more general terms, then, the main part of your program should include statements that poll the event queue on a regular basis and act on any relevant event. To do this, you must design that main part as one large poll-and-respond loop. In Apple IIGS terminology, this is the *event loop*.

The Event Loop

The event loop must contain statements that do three things:

1. Read the event queue to find out whether an event has taken place.
2. Pull an event from the queue and determine its type (e.g., key press or mouse).
3. Perform the action the event signifies.

Of course, among the events that the program must check for is one indicating that the user wants to quit. This usually involves selecting “Quit” from a menu. When that happens, the program must exit the event loop, shut down the tools, and return to the calling program.

Until the user says to quit, however, the statements in the loop must be repeated indefinitely. Thus, while an “event-driven” program may seem to be simply waiting for something to happen, it’s actually racing through the event loop time and again, and polling the event queue during each pass.

Event Types

The Event Manager can keep track of seven types of events, plus a nonevent that indicates the queue is empty; in other words, no event has taken place. The Event Manager keeps information about each event, classifying it as to type.

Mouse Events

When someone presses the mouse button, the Event Manager posts it in the event queue as a *mouse-down event*. Similarly, releasing the mouse button gets posted as a *mouse-up event*. Generally, a mouse-up event is only significant for detecting the end of a dragging operation; for most simple applications, only mouse-down events are meaningful.

For both types of mouse events, the Event Manager also records the screen coordinates of the mouse pointer when the event occurred, because they may indicate something to your program. The Event Manager provides a `GetMouse` tool call that returns the mouse pointer coordinates.

Although the standard Apple IIGS mouse only has one button, some joysticks have two. For this reason, the Event Manager provides for two separate buttons, which it numbers 0 and 1 (where 0 is the button on the mouse).

Keyboard Events

When a user presses a letter, number, or symbol key on the main keyboard or the keypad, the Event Manager records it as a *key-down event*. Shift, Control, Caps Lock, Option, and Open-Apple (which are called “modifier” keys) are treated differently. Pressing them does not produce an event, but if you press one at the same time as a character key, the Event Manager reports that fact in the key’s event record. Knowing the state of the modifier key lets you distinguish uppercase and lowercase characters; for example, it lets you differentiate an *m* (only the M key was pressed) from an *M* (both Shift and M were pressed).

If you hold a character key down long enough to make it repeat, the Event Manager records that as an *auto-key event*. Thereafter, it records an additional auto-key event each time the key actually repeats. The Options selection in the Control Panel provides two parameters that let you change the keyboard’s repeat characteristics. Repeat Delay controls the amount of time it takes for a key to start repeating, while Repeat Speed controls the repetition rate.

Window Events

Window events are produced (as you may have guessed) by the Window Manager. There are two kinds of window events, activate and update.

An *activate event* results from a user activating or deactivating windows that overlap on the screen. Clicking the mouse pointer inside a partially covered window (to make it active) constitutes one activate event, and the currently active window becoming inactive constitutes a separate activate event. Thus, activate events generally occur in pairs.

Because an activate event signifies an action that must be performed as soon as possible — that is, activate or deactivate a window — the Event Manager doesn’t even post it in the event queue. Instead, the Event Manager constructs an event record for it and notes its occurrence at the top of a priority list (discussed shortly). When your program next asks for an event from the event queue, the Event Manager says, “Forget the event queue, my friend, I have something more important”, and hands the activate event to the program.

The other type of window event, called an *update event*, generally follows an activate event. With an update event, the Window Manager tells the application program (via the Control Manager) that one or more windows on the screen must be redrawn — usually because the user has opened, closed, activated, or moved a window. The Window Manager does the actual drawing, as you shall see later, but it takes its “get started” cue from the Control Manager.

As with activate events, the Event Manager creates an event record for each update event, but does not post its occurrence in the event queue. Update events are placed at the bottom of the Event Manager's priority list, however; even events in the queue are deemed more significant.

Switch Events

Just as the Window Manager generates window events, the Control Manager generates *switch events*. It produces a switch event when the user presses the mouse button (or sometimes a key) to indicate that he or she wants to switch from this application to another. In effect, a switch event tells the Control Manager to inform the program that it should do whatever housekeeping is necessary (e.g., update the screen and save critical information) before switching to the new application.

Like window events, switch events produce only an event record and are not entered in the event queue. However, since switching applications is such a drastic action, the Event Manager puts it second on the priority list, just below activate events.

User-Defined Events

The Apple IIGS ROM contains all the necessary firmware for communicating with standard peripheral devices, such as printers and modems. However, if some nonstandard device (say, an electronic piggy bank) is connected to your computer, you must write a *device driver* program to communicate with it. Input from such devices should be posted to the event queue as *device driver events*, using the PostEvent tool call.

The Event Manager can also accommodate up to four other general-purpose events that you can use. Your program should post them to the event queue as *application events*, using the PostEvent call.

Desk Accessory Events

A *desk accessory event* is generated when you press the OpenApple-Control-Esc key combination. Pressing these keys brings on the classic desk accessory menu, the one you use to obtain the Control Panel. You can ignore desk accessory events, because your program will never receive them; they are intercepted and handled by the Desk Manager.

The Null Event

Once your program retrieves an event, it should do whatever that event signifies (if anything), then go back and ask the Event Manager for the next

event. It should continue retrieving and processing events until the Event Manager returns a *null event*. This indicates that there are no more events to process, either in or out of the event queue. Indeed, null events are the best kind of all!

Event Priorities

As I mentioned in the preceding section, the Event Manager posts most events in the queue, but it places activate and update events from the Window Manager, and switch events from the Control Manager, in a priority list. When your program polls the Event Manager with a *GetNextEvent* call (described later), it returns the event record of the event that has the highest priority in its list. In order of decreasing importance, the priorities are:

1. Activate events (a window becoming active or inactive).
2. Switch events (the user wants to switch applications).
3. Mouse-down, mouse-up, key-down, auto-key, device driver, application-defined, and desk accessory events. The Event Manager posts all of these events in the event queue.
4. Update events, in front-to-back window order.
5. Null event.

Thus, when you tell the Event Manager to *GetNextEvent*, it begins by checking for a pending activate event. If one is available, it is passed to your application. Because of the way activate events are generated, there can never be more than two pending at the same time — one for a window being activated and the other for a different window being deactivated.

If no activate events are pending, the Event Manager looks for a pending switch event. Since a switch event signifies that the user wants to switch applications, the Event Manager temporarily forgets about priorities and digs down to the bottom of the list to see if any update event is pending. If an update is available, the Event Manager passes *it* (rather than the switch event) to your program. This signals the program to update all the windows before the switch takes place. That way, the windows will be intact if the user later returns to this application.

In the absence of pending activate and switch events, the Event Manager looks into the event queue. It always posts events in the queue in the order

it detects them. It also removes events *from* the queue in the same order, with the “oldest” event being removed first. Hence, the event queue operates like a vending machine; the package that was loaded into the machine (queue) first will be the one dispensed when you push the button (call `GetNextEvent`).

In technical terms, a queue is a data structure that operates in “first-in/first-out” fashion. Compare this to a data structure you encountered earlier, the 65816’s stack, which operates in “last-in/first-out” fashion.

When the event queue has been emptied, the Event Manager looks for a pending update event. It should find one if an activate event ever occurred — provided, of course, that the user has not requested an application switch in the meantime. (Switch events deal with pending update events automatically.) Finally, if no events are pending, the Event Manager passes a null event to the application.

Event Records

Whenever the Event Manager detects an event of any kind (queue or non-queue), it creates an *event record* that contains all the pertinent information about that particular event. It is this record that your program receives when it issues a `GetNextEvent` call. An event record contains five fields:

What	word	(event code)
Message	long integer	(event message)
When	long integer	(elapsed time since start-up)
Where	point	(mouse location)
Modifiers	integer	(modifier flags)

What — Event Codes

The *What* field holds a number from 0 to 15 that identifies the type of event that occurred (see Table 8-1). Your program can use this number to call the routine associated with that particular kind of event (or continue polling, if it’s the null event).

In practice, your event loop should use the event code to obtain the address of the appropriate event routine from a table, then execute that routine. Example 8-1 illustrates this approach.

This particular event loop accepts only mouse-down and key-down events, and calls `DoMouseDown` or `DoKeyDown` (not shown here) to

Table 8-1

0	- Null event
1	- Mouse-down
2	- Mouse-up
3	- Key-down
4	- Undefined
5	- Auto-key
6	- Update
7	- Undefined
8	- Activate
9	- Switch
10	- Desk accessory
11	- Device driver
12	- Application-defined
13	- Application-defined
14	- Application-defined
15	- Application-defined

Example 8-1

```

EventLoop  START
            using GlobalData

Again      anop                                ;Beginning of event loop
            lda QuitFlag                      ;Does user want to quit?
            bne AllDone                       ; If so, branch to AllDone
            ..                                (Otherwise, get next event record)
            ..
            lda EventWhat                     ;Get event code from event record,
            asl a                             ; double it,
            tax                               ; and put it in X

            jsr (EventTable,x)                ;Execute the event's routine
            bra Again

AllDone    rts

EventTable anop                                ;Event Manager events
            dc i'Ignore'                      ; 0 null
            dc i'JoMouseDown'                 ; 1 mouse-down
            dc i'Ignore'                      ; 2 mouse-up
            dc i'DoKeyDown'                   ; 3 key-down
            dc i'Ignore'                      ; 4 undefined
            ..                                (etc.)
            ..
            END

Ignore     START
            rts
            END

```

process them. All other events send the program to the Ignore subroutine, which has only one instruction, RTS. Note that because the addresses in EventTable are 2 bytes long, the event code must be doubled (by shifting it left one bit position) before it can be used as an index.

Message — Event Message

The *Message* field of an event record contains additional information about the event. The actual contents of the event message depends on what type of event it is, as follows:

Event Type	Event Message
Key-down	ASCII character code in low byte
Auto-key	ASCII character code in low byte
Mouse-down	Button number (0 or 1) in low byte
Mouse-up	Button number (0 or 1) in low byte
Activate	Pointer to window that generated event
Update	Pointer to window that needs redrawing
Device driver	User-defined
Application	User-defined
Switch	Undefined
Desk Accessory	Undefined
Null	Undefined

When — Elapsed Time

The event record's *When* field contains the elapsed time since you started the computer, in sixtieths of a second, or "ticks." This is handy for determining the order in which events occurred, in case you're interested.

Where — Mouse Pointer Location

The *Where* field gives the screen coordinates of the mouse pointer when the event occurred. These coordinates are meaningful for mouse-up events, because the Control Manager can use them to determine whether the pointer was in a significant area when the user released the mouse button.

Modifiers — Modifier Flags

The final item in an event record, *Modifiers*, is a word-size unit whose bits reflect the state of the modifier keys and certain other conditions when the event occurred. Figure 8-1 shows the arrangement of the bits in this *modifier*

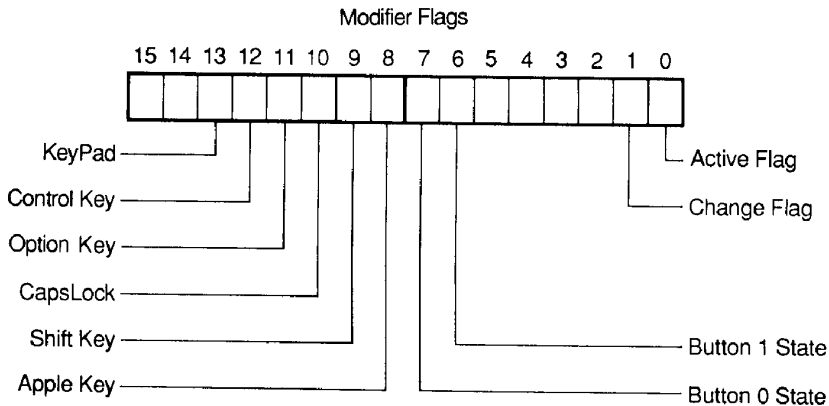


Figure 8-1

flags word. For each flag, a 1 indicates “true” (e.g., key is being pressed) and a 0 indicates “false” (e.g., key is unpressed).

Here, the upper byte (bits 8 through 15) reflects key events, while the lower byte (bits 0 through 7) reflects mouse and activate events. Control, Option, Caps Lock, Shift, and Apple are, of course, the modifier keys on the main keyboard. The Keypad flag (bit 13) is 1 if the user has pressed a key on the numeric keypad.

Button 0 State and Button 1 State track the two possible buttons on a controller; the button on the regular Apple IIGS is numbered 0. These bits behave the opposite of their keyboard counterparts; that is, a button bit set to 1 indicates that the button was in the *up* (unpressed) position, while a 0 indicates that it was *down* (pressed). Therefore, a mouse-down event will have the Button 0 State bit set to 0 and the Button 1 State bit set to 1.

Normally, the modifier flags are checked by an event routine; a subroutine that has been called from the program’s event loop. For example, suppose a key-down event has occurred, and you want to check whether the user has pressed OpenApple as well as the character key (perhaps to select from an on-screen menu). In response to a key-down event, the event loop calls, say, a *DoKeyDown* subroutine. To check for the OpenApple key, *DoKeyDown* would test whether bit 8 of the modifier flags is 1 (OpenApple down) or 0 (OpenApple up), and act accordingly. Thus, *DoKeyDown* may include this kind of instruction sequence:


```

        lda EventModifiers      ;Read modifier flags
        and #$100              ;Zero all but Open-
                                ; Apple key bit
        bne AppleDown          ;Is OpenApple key
                                ; pressed?
        . .
        . .
AppleDown . .                  ;Yes.
        . .

```

Event Manager Tool Calls

Table 8-2 summarizes the most common tool calls to the Event Manager.

Table 8-2

Housekeeping Calls	
<u>__EMStartup</u>	Start the Event Manager
Call with: PushWord <i>DirectPageLoc</i>	;Loc. of 1-page work area
PushWord <i>QueueSize</i>	;Max. number of queue events
PushWord <i>XMinClamp</i>	;Leftmost column for mouse
PushWord <i>XMaxClamp</i>	;Rightmost column for mouse
PushWord <i>YMinClamp</i>	;Top row for mouse
PushWord <i>YMaxClamp</i>	;Bottom row for mouse
PushWord <i>ProgramID</i>	;The program's ID number
<u>__EMStartup</u>	
Result: None	
Note: See text for descriptions of the inputs.	
<u>__EMShutDown</u>	Shut down the Event Manager
Call with: <u>__EMShutDown</u>	
Result: None	
Calls That Access Events	
<u>__GetNextEvent</u>	Get the next event
Call with: PushWord #0	;Space for result
PushWord <i>EventMask</i>	;Event mask
PushLong <i>EventPtr</i>	;Pointer to event record
<u>__GetNextEvent</u>	
Result: Word containing event code	
Note: See text for description of EventMask. EventPtr points to a buffer of the form:	
EventRecord anop	

Table 8-2 (cont.)

Calls that Access Events (cont.)	
EventWhat	ds 2
EventMessage	ds 4
EventWhen	ds 4
EventWhere	ds 4
EventModifiers	ds 2

__FlushEvents	Remove specified events from the event queue
Call with: PushWord #0	;Space for result
PushWord <i>EventMask</i>	;Event mask
PushWord <i>StopMask</i>	;Stop mask
__FlushEvents	
Result:	Word value. If all events were removed, the word contains 0. If an event caused the removal operation to stop, the word contains its event code.
Note:	See text for descriptions of EventMask and StopMask.

Mouse-Reading Calls	
__GetMouse	Read the current mouse location
Call with: PushLong <i>MouseLocPtr</i>	;Pointer to a point buffer
__GetMouse	
Result:	None
Note:	MouseLocPtr points to a buffer of the form:
ds 2	;Row
ds 2	;Column

__Button	Test the up/down state of the mouse button
Call with: PushWord #0	;Space for result
PushWord <i>ButtonNum</i>	;Button number (0 or 1)
__Button	
Result:	1 if button is down, 0 if it is up.

Housekeeping Calls

The EMStartup call has some unusual-sounding inputs that are worth discussing. At the end of this section, I will tie everything together by listing the standard EMStartup sequence.

The Event Manager requires one page of working space in bank 0, and you obtain it (along with space for other tool sets) by issuing a NewHandle call to the Memory Manager. In the generalized program model (Example 6-2), the Event Manager's space follows the three pages that QuickDraw requires. Hence, that's the address to which *DirectPageLoc* should point.

The *QueueSize* input specifies the maximum number of events the queue can hold. A value of 0 sets up a default size of 20 events.

The *Clamp* inputs specify the area of the screen where the mouse position is meaningful. To make the mouse active throughout the entire screen, you would set *YMinClamp* to 0, *YMaxClamp* to 200, *XMinClamp* to 0, and *XMaxClamp* to the rightmost column — 320 for 320 mode or 640 for 640 mode.

Just as the program model employs a constant called *ScreenMode* to hold the master scan line control byte (SCB) value, it employs another constant, *MaxX*, to hold the *XMaxClamp* value. Constants are handy here, because they let you configure a new program for either 320 mode or 640 mode by changing only these two values in the model.

The final input, *ProgramID*, is the program identification number returned by the Memory Manager *MMStartup* call. The program model stores it in a variable called *MyID*.

The program model in example 6-2 uses the following sequence to start the Event Manager:

```

lda 4           ;Starting address of reserved space
clc            ;ZP to use = QD ZP + $300
adc # $300
pha           ;Push DirectPageLoc input
PushWord #20   ;Queue should accept 20 events
PushWord #0    ;Column clamp left
PushWord #MaxX ;Column clamp right
PushWord #0    ;Row clamp top
PushWord #200  ;Row clamp bottom
PushWord MyID  ;Program ID
__EMStartup
ldx #6
jsr PrepareToDie

```

Calls that Access Events

The *GetNextEvent* call copies the event record of the next available event into the buffer to which *EventPtr* points. The *EventMask* input is a word-size value that lets you specify the type(s) of events your program can process; Figure 8-2 shows its format. Here, a 1 in a bit position tells the Event Manager to pass events of that type to your application, while a 0 tells it to ignore those events. Thus, a program that's waiting only for a key press from the user would probably want to accept only key-downs. In this case, you would push an event mask value of 8 (binary 1000) onto the stack.

GetNextEvent returns a true/false indicator that contains a nonzero

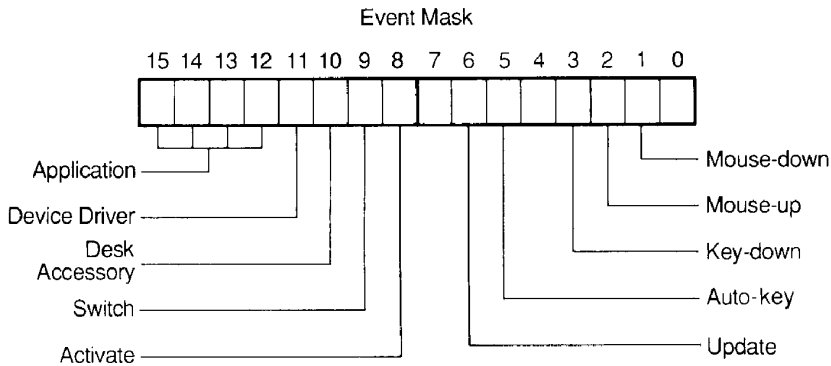


Figure 8-2

value if an event satisfies your event mask criteria or zero if no such events occurred. A nonzero value in the indicator signals your program to use the event code to call the routine that processes events of that type.

To illustrate, Example 8-2 shows a portion of an event loop for a program that accepts only key-down and auto-key events. Each key-down makes the program execute `DoKeyDown`; each auto-key makes it execute `DoAutoKey`.

`DoKeyDown` may do any of several things. If the key character (in the event record's Message field) represents text, `DoKeyDown` will simply display it on the screen. If the character indicates an option the user has selected from an on-screen menu, `DoKeyDown` will execute the routine that services that option — provided, of course, the character represents a valid menu choice.

It's important to realize that `GetNextEvent`'s event mask simply tells the Event Manager which types of events to pass to the application. Event types that are masked out (with a 0 in the event bit) still exist; the Event Manager still has records for them and queue-related events remain in the queue. This means that by using a mask, you run the risk of eventually filling up the queue. When the queue is full and the Event Manager detects a new event, it simply discards the earliest event in the queue — and that may well be an event your program needs!

There are several ways to guard against losing relevant events. The easiest way is to accept *every* event (using a mask of all 1's, or \$FFFF) and let your event table do the masking. In this case, the event table should contain a subroutine address for each event you want to accept and the address

Example 8-2

```

EventLoop  START
            using GlobalData
Again      anop                ;Beginning of event loop

            lda QuitFlag        ;Does user want to quit?
            bne AllDone         ; If so, branch to AllDone

            PushWord #0         ;Space for result
            PushWord ##28       ;Accept only key events
            PushLong #EventRecord ;Point to event record buffer
            _GetNextEvent

            pla                 ;Any key events?
            beq Again           ; No. Continue polling
            lda EventWhat       ; Yes. Read event code into A,
            asl a               ; double it,
            tax                 ; and put it in X

            jsr (EventTable,x)  ;Execute the event's routine,
            bra Again          ; then continue polling
AllDone    rts

EventTable anop                ;Event Manager events
            dc i'Ignore'        ; 0 null
            dc i'Ignore'        ; 1 mouse-down
            dc i'Ignore'        ; 2 mouse-up
            dc i'DoKeyDown'     ; 3 key-down
            dc i'Ignore'        ; 4 undefined
            dc i'DoAutoKey'     ; 5 auto key
            ..                  (etc.)
            ..
            END

Ignore     START
            rts
            END

DoKeyDown  START
            ..                  ;This routine is executed if
            ..                  ; a key-down event occurs
            rts
            END

DoAutoKey  START
            ..                  ;This routine is executed if
            ..                  ; an auto-key event occurs
            rts
            END

```

of a do-nothing “ignore” subroutine for all other entries, as in Example 8-2. This technique empties the queue automatically (provided you poll often enough to keep up with events), but it can be impractical for some applications.

Consider, for instance, an application in which only keystrokes are relevant in one part of the program, only mouse clicks are relevant in another part, and both keys and the mouse are acceptable in a third part. Here, masking with the event table would require the program to monitor the context in which each event occurred. You could instead make the event mask block out unwanted events and call *FlushEvents* to empty the queue when all required events have been received.

FlushEvents removes events from the queue based on two separate event masks. Specifically, it removes all events of the types marked 1 in *EventMask* up to (but excluding) the first event of any type marked 1 in *StopMask*. To clear the queue of all types specified by *EventMask*, use a *StopMask* value of 0. To clear the queue entirely, without regard to type, set *EventMask* to \$FFFF and *StopMask* to 0.

Mouse-Reading Calls

The two most useful mouse-related tool calls are *GetMouse* and *Button*. *GetMouse* reads the current mouse location into the record to which *MouseLocPtr* points. This location is given in the “local” coordinate system of the current *GrafPort*, which might be a window that occupies just a portion of the screen. This differs from the mouse location in the *Where* field of an event record, which gives the mouse’s “global” (screen) coordinates.

The *Button* call checks either of two buttons (0 or 1) on a controller and returns 1 if it is down (pressed) or 0 if it is up (unpressed). Recall that 0 is the button number for the standard Apple IIGS mouse.

A Simple Program that Uses the Event Manager

You will employ the resources of the Event Manager in virtually every program you write, so it’s time to look at a real program that uses it. Recall that the example programs in Chapter 7 use an Apple II-style routine that lets you exit a program by pressing any key. This is certainly a candidate for replacement by Event Manager calls.

However, instead of just adding a few minor embellishments to an old program, I will present a program that displays text as you type it, and quits when you press the Esc key. Admittedly, this won’t be a very elegant program, but it should tie this chapter together. So take a moment to examine Example 8-3, the listing of a program I call *SHOWTEXT*.

SHOWTEXT’s event loop calls *GetNextEvent* repeatedly until it receives a nonzero value from the stack. When that happens, *EventLoop*

Example 8-3

; SHOWTEXT displays text from the keyboard until the user presses Esc.

```

        absaddr on
        MCOPY showtext.macros

ShowText      START
              using GlobalData

;-----
;
; Global equates used throughout the program.

ScreenMode    gequ $80                ;640 mode, no fill
MaxX          gequ 640                ;640 mode for Event Manager

              phk                      ;Set data bank to program
              plb                      ; bank to allow absolute addressing

              jsr InitStuff            ;Initialize everything
              bcs AllDone              ;Quit if initialization fails

              stz QuitFlag             ;Initialize the quit flag to 0
              jsr EventLoop            ;Display the user's text

AllDone       anop                    ;All is done, shut down
              _DeskShutDown            ;Desk Manager
              _EMShutDown              ;Event Manager
              _QDShutDown              ;QuickDraw II
              _MTShutDown              ;Miscellaneous Tools

              PushWord MyID             ;Discard the program's handle
              _DisposeAll

              PushWord MyID             ;Memory Manager
              _MMShutDown              ;Tool Locator

              _Quit QuitParams          ;Do a ProDOS Quit call
              brk $F0                  ;If it fails, break

              END

;*****
;
; Global Data
;
;*****

GlobalData    DATA

QuitParams    dc i4'0'                ;Return to caller
              dc i'$4000'              ;Make program restartable in memory

ToolTable     dc i'NumTools'           ;No. of tool sets in table
              dc i'4,$0100'            ;QuickDraw
              dc i'5,$0100'            ;Desk Manager
              dc i'6,$0100'            ;Event Manager

TTEnd         anop

```

288 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

```

TableSize      equ TTEnd-ToolTable-2
NumTools       equ TableSize/4

EventRecord     anop                      ;Buffer for event record
EventWhat       ds 2
EventMessage    ds 4
EventWhen       ds 4
EventWhere      ds 4
EventModifiers  ds 2

MyID            ds 2                      ;This will hold the program's i.d.

VolNotFound     equ $45                  ;ProDOS error

QuitFlag        ds 2                      ;Quit flag
END

;*****
;
; InitStuff
;
; Gets space in bank zero for use as direct page by tools and
; then initializes QuickDraw and the Event Manager.
;
;*****

InitStuff       START
                using GlobalData

; Initialize the Tool Locator, Memory Manager, and Miscellaneous
; Tools.

                _TLStartup

                PushWord #0
                _MMStartup

                pla                      ;Memory Manager returns program's ID
                sta MyID

                _MTStartup              ;Misc. tools
                ldx #3
                jsr PrepareToDie

; Get some space for the direct page we need. QuickDraw needs
; three pages and the Event Manager needs one page.

                PushLong #0             ;Space for handle
                PushLong #$400          ;Four pages
                PushWord MyID           ;Owner
                PushWord #$C005         ;Locked, fixed, fixed bank
                PushLong #0             ;Location
                _NewHandle
                ldx #$FF
                jsr PrepareToDie

                pla                      ;Read handle and store in dp
                sta 0
                pla
                sta 2

```



```

        lda [0]                ;Get dp location from handle
        sta 4                  ; and store at loc 4 of dp

; Initialize QuickDraw

        pha                    ;Use dp obtained from handle
        PushWord #ScreenMode   ;Mode = 640
        PushWord #0            ;Use entire screen
        PushWord MyID
        _QDStartup
        ldx #4
        jsr PrepareToDie

; Initialize Event Manager

        lda 4                  ;dp to use = QD dp + $300
        clc
        adc #$300
        pha
        PushWord #20           ;Queue size
        PushWord #0            ;X clamp left
        PushWord #MaxX         ;X clamp right
        PushWord #0            ;Y clamp top
        PushWord #200          ;Y clamp bottom
        PushWord MyID
        _EMStartup
        ldx #6
        jsr PrepareToDie

; -----
;
; Load the RAM-based tools I need

LoadAgain    PushLong #ToolTable
              _LoadTools
              bcc ToolsLoaded

              cmp #VolNotFound
              beq DoMount
              sec
              ldx #$FE
              jsr PrepareToDie

DoMount      anop
              jsr MountBootDisk
              cmp #1
              beq LoadAgain

              sec
              rts

; The tools have been loaded. Initialize the Desk Manager.

ToolsLoaded  anop
              _DeskStartup

              clc                ;Clear the carry flag
              rts                ; and return

END

```

```

;*****
;
; Event Loop
;
; Display keyboard characters until user presses Esc.
;
;*****

EventLoop      START
                using GlobalData

                PushWord #20                ;Start the pen at (20,20)
                PushWord #20
                _MoveTo

Again          lda QuitFlag
                bne AllDone

                PushWord #0                ;Space for result
                PushWord #$FFFF            ;Accept any event
                PushLong #EventRecord      ;Point to event record buffer
                _GetNextEvent

                pla                        ;Is an event available?
                beq Again                  ; No. Continue polling
                lda EventWhat              ; Yes. Read event code into A,
                asl a                      ; double it,
                tax                        ; and copy it into X.

                jsr (EventTable,x)         ;Execute the event's routine,
                bra Again                  ; then resume polling

AllDone        rts

EventTable      anop                      ;Event Manager events
                dc i'Ignore'              ; 0 null
                dc i'Ignore'              ; 1 mouse down
                dc i'Ignore'              ; 2 mouse up
                dc i'DoKeyDown'            ; 3 key down
                dc i'Ignore'              ; 4 undefined
                dc i'DoAutoKey'            ; 5 auto-key
                dc i'Ignore'              ; 6 update
                dc i'Ignore'              ; 7 undefined
                dc i'Ignore'              ; 8 activate
                dc i'Ignore'              ; 9 switch
                dc i'Ignore'              ; 10 desk acc
                dc i'Ignore'              ; 11 device driver
                dc i'Ignore'              ; 12 ap
                dc i'Ignore'              ; 13 ap
                dc i'Ignore'              ; 14 ap
                dc i'Ignore'              ; 15 ap
                END

;*****
;
; Ignore
;
; This do-nothing subroutine returns to the event loop for
; events we don't want.
;
;*****

```

```
Ignore    START
          rts
          END
```

```
;*****
;
; DoKeyDown
;
; If user pressed Esc, DoKeyDown sets the QuitFlag to 1. It sends
; other keys to the screen, provided they are displayable.
;
;*****
```

```
DoKeyDown    START
              using GlobalData

              lda EventMessage      ;Read the ASCII character
              cmp #27               ;Is it Esc?
              bne NotEsc

              sta QuitFlag          ; Yes. Quit
              rts

NotEsc        anop                  ; No. See if it's displayable
              cmp #$20              ; (in the range $20 to $7E)
              bcc Exit
              cmp #$7F
              beq Exit

              pha
              _DrawChar              ;Display the character

Exit          rts
              END
```

```
;*****
;
; DoAutoKey
;
; Display this key on the screen.
;
;*****
```

```
DoAutoKey    START
              using GlobalData

              lda EventMessage      ;Read the ASCII character
              cmp #$20              ;To be displayed, a character
              bcc Leave             ; must be in the range $20
              cmp #$7F              ; to $7E
              beq Leave

              pha
              _DrawChar              ;Display the character
              rts

Leave          rts
              END
```

292 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

```
*****
*
* Prepare To Die
*
* Dies if carry is set.
*
* Error number to display is in X register.
* Assumes that Miscellaneous Tools is active.
*
*****

PrepareToDie  START
               bcs RealDeath      ;Carry = 1?
               rts                 ; No. Return to caller

RealDeath     phx                   ; Yes. Goodbye, program.
               PushLong #DeathMsg
               _SysFailMgr

DeathMsg      str 'Could not handle error '

               END

;*****
;
; Mount Boot Disk
;
; Tells the user to insert the disk that contains the RAM-based
; tools.
;
;*****

MountBootDisk START
               _Set_Prefix SetPrefixParams
               _Get_Prefix GetPrefixParams

               PushWord #0           ;Space for result
               PushWord #195         ;Column position for dialog box
               PushWord #30          ;Row position for dialog box
               PushLong #PromptStr   ;Prompt at top of dialog box
               PushLong #VolStr      ;Volume name string
               PushLong #OKStr       ;String in Button 1
               PushLong #CancelStr   ;String in Button 2
               _TLMountVolume

               pla                   ;Obtain the button number
               rts                   ; and return to caller

PromptStr     str 'Please insert the disk.'
VolStr        ds 16
OKStr         str 'OK'
CancelStr     str 'Shutdown'

GetPrefixParams dc i'7'
               dc i4'VolStr'
SetPrefixParams dc i'7'
               dc i4'BootStr'

BootStr       str '*/'

               END
```

uses the event code to obtain a subroutine address from an event table. Only two kinds of events are meaningful here: key-downs (which call `DoKeyDown`) and auto-keys (which call `DoAutoKey`).

`DoKeyDown` reads the key's ASCII character code from the event record's Message field and sets a quit flag if the code is 27, or Esc. (See Appendix B for the ASCII character codes.) If the key is something other than Esc, `DoKeyDown` tests whether the code is displayable (i.e., whether it's between \$20 and \$7E) and, if so, displays it with a QuickDraw `DrawChar` call.

`DoAutoKey` is identical to `DoKeyDown`, except it does not check for Esc.

Again, `SHOWTEXT` is not very elegant, and you could probably improve it. For example, it doesn't detect the end of a line; it simply keeps sending characters to the screen, without accounting for the fact that you can't see them. To eliminate this problem, you could follow each character-display operation with a QuickDraw `GetPen` call, and move the pen to the next line if the column position is equal to or greater than the rightmost column of the screen (column 319 or 639).

You could also modify the program to accept the Tab and Return keys, where Tab moves the pen to the next predefined tab stop and Return moves it to the beginning of the next line. In fact, by adding enough enhancements, you could actually expand `SHOWTEXT` into a full-fledged word processor! As textbooks often say, "This exercise is left to the reader."

What's Next?

The programs in this chapter and the preceding one illustrate the techniques you should use to communicate with Apple IIGS tool sets and show some rudimentary ways to display graphics and process events. However, none of these programs provide meaningful interaction with the user. Except for `SHOWTEXT`, the user can only view what the program puts on the screen, and press a key when he or she wants to quit. Surely, you will want to be able to do more than that!

For example, you may want to show portions of several pictures on the screen, in overlapping fashion, and let the user choose the one he or she wants to see. You may also want your program to present a menu upon demand, to let the user indicate with the mouse what should be done next. To do these kinds of tasks, you need the services of tool sets I have not yet described. Let's take them one at a time, starting with the Window Manager.

CHAPTER 9

Working with Windows

The programs presented thus far work with only a single object — a graphics picture or a block of text — on the screen. However, sometimes you may want to let the user select from several different objects. To deal with multiple objects, either pictures or text, you must put them in individual “windows” — and for that you need the services of the *Window Manager*.

A window is an object on the screen that contains graphics or text, or both. Usually a window shows an entire data area (e.g., an entire picture), but if the data area is too large to fit on the screen, the window can show just a portion of it. Thus, a window can represent the user’s view of a larger object, just as a window in your home or office is your view of a *much* larger object: the world outside!

The Apple IIGS provides two types of standard windows, called *document* and *alert* (see Figure 9-1). Alert windows are produced by the Dialog Manager; they are described in Chapter 12. The Window Manager displays document windows, so that’s the kind that interests us at the moment. To save you from having to read “document windows” throughout the rest of this chapter, I will simply refer to them as “windows.”

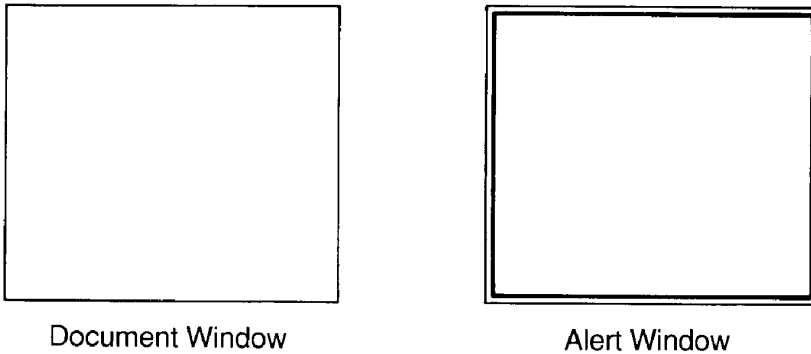


Figure 9-1

Window Components

You can make a window as plain or elaborate as you want, and the Window Manager offers a variety of standard components you can add to windows. Some components are *controls* that let the user operate on the information being displayed, or on the window itself, by clicking the mouse button. Other components are informational. Figure 9-2 shows a window that's equipped with all the predefined (or standard) components.

Content Region

Every window has a content region. It is the window's main area, the place where it displays your graphics or text. That will be the entire window if you haven't added other components.

Title Bar

The title bar displays the window's name or title. It is also a control region; the user can move the window — perhaps to reveal a window beneath it — by simply dragging it by its title bar.

Close and Zoom Boxes

The title bar can include a close box at the left or a zoom box at the right, or both. Clicking the mouse in a *close box* makes the window retreat into an icon or simply disappear. It's up to your application program to control what

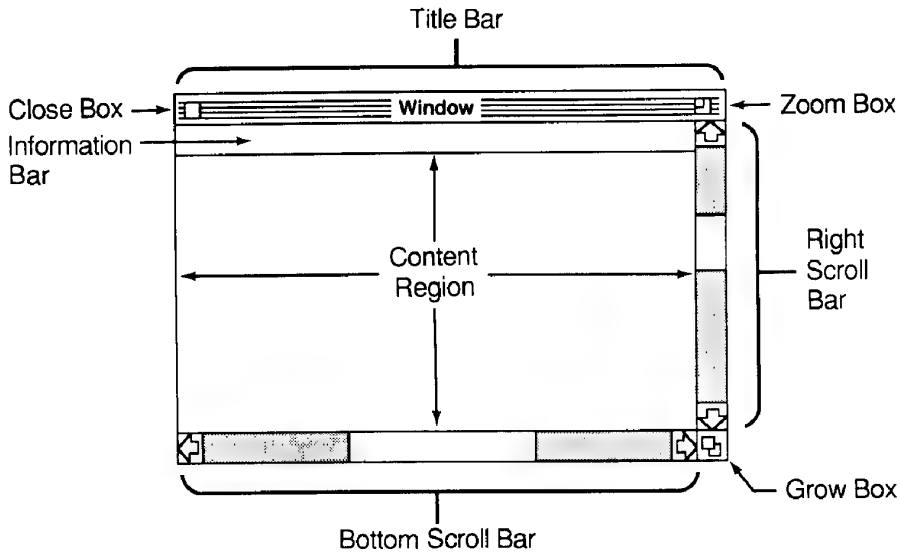


Figure 9-2

happens when a window is closed, but in general, you should simply hide the window (make it invisible), so it can be reopened later.

The application program determines the initial size of a window and its position on the screen. As just mentioned, however, you can change a window's position by dragging it by the title bar. Another control, the *zoom box*, lets you change a window's size. (A grow box also lets you resize a window, but is a more versatile control than a zoom box — more about that shortly.)

By clicking in the zoom box, the user can expand a window to some maximum size that's been preset by the program. (Most application programs make windows zoom to fill the entire screen.) Once a window has been zoomed to its maximum size, clicking in the zoom box again shrinks the window back to its previous size and position.

Grow Box

While the zoom box is like an on/off switch that produces only two window sizes (original size or full size), the grow box lets you expand or shrink a window incrementally, to whatever size you want. To resize a window, put the mouse pointer in its grow box and drag the box to where you want it to end up. Moving away from the window makes it expand and show more of

the pixel image; moving toward the window, or into it, makes it contract and show less of the image.

Once you have resized a window using the grow box, it retains that size until you drag the grow box again. Thus, while zooming a “grown” window still expands it to full size, zooming a full-size window shrinks it to its *grown* size — not to its initial size.

Scroll Bars

If your picture is too large to fit on the screen, you can display just a portion of it in a window and provide scroll bars to let the user work his or her way through the rest of the data area. A scroll bar has arrows at each end and a white rectangular “scroll box” in the gray area between the arrows.

Clicking in one of the right scroll bar’s arrows “moves” the data area upward or downward beneath the window. Similarly, clicking in the bottom scroll bar’s arrows moves it to the right or left. (In each case, the number of rows or columns the data area actually moves must be set by the application program.)

Thus, scroll bar arrows function like the control knobs on a microfiche viewer. Just as you rotate the “vertical” knob on a viewer to make a higher or lower portion of the fiche appear on the screen, you click the right scroll bar’s top or bottom arrow to move the data area upward or downward beneath the window’s content region. I could also compare a viewer’s “horizontal” knob to the bottom scroll bar’s left and right arrows, but you get the idea.

The *scroll box* or *thumb* within each scroll bar indicates the current position of the content region within the larger data area. That is, the scroll boxes show how far down or to the right the content region is from the beginning of the data area (see Figure 9-3). As the user “moves” through the data area by clicking a scroll arrow, the scroll box moves vertically or horizontally to reflect the change in position.

The size of a scroll box is also meaningful insofar as it indicates how high or wide the data area actually is. If the data area is very high (i.e., long), the right scroll box will be small; it will cover only a small portion of the gray area. Conversely, if most of the data area is already on the screen, the right scroll box will cover most of the gray area. Thus, the scroll boxes not only indicate the relative position of the content region, but also how much of the data area is being displayed.

Besides being position and size indicators, the scroll boxes are also movement controls; they do the same thing as the scroll bar arrows, but

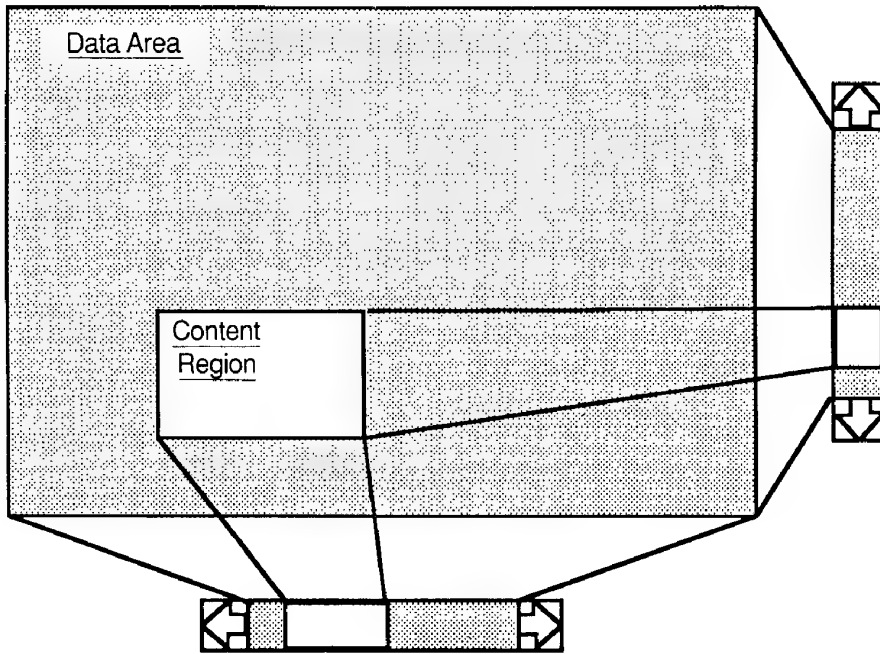


Figure 9-3

faster. To move vertically or horizontally through a data area, simply drag the scroll box up or down, left or right, to where you want it. The instant you release the mouse button, the content region will change to show the portion of the data area you have selected.

The gray area in a scroll bar is also a movement control; clicking in it moves the data area by a “page” worth of pixels. By convention, this means the data area moves ten times as far as it does when you click in a scroll bar arrow. Thus, if clicking the up arrow in the right scroll bar displays the next four rows of data, clicking in the top gray area displays the next 40 rows.

In short, there are three ways to move a data area under a window: by clicking in an arrow to scroll incrementally, by clicking in the gray area to “page” a window at a time, and by dragging the scroll bar a selected distance. Figure 9-4 illustrates these three techniques.

Information Bar

The information bar is an optional strip of space below the title bar that your program can use to display additional graphics or text. You could use it to

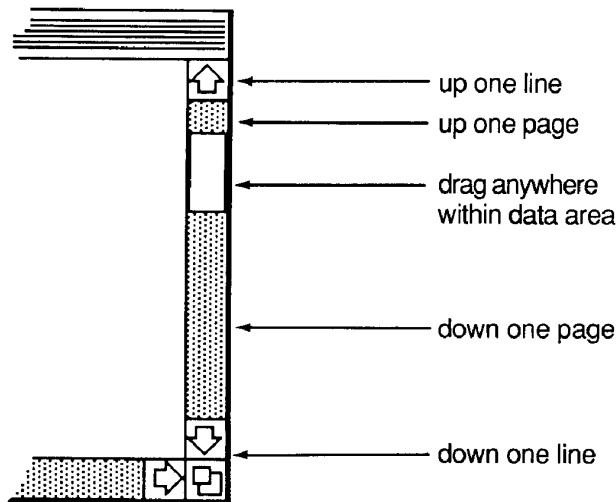


Figure 9-4

display something simple, such as messages to the user, or something more complex, such as the “scoreboard” in a computerized football game, a treasure map in an adventure game, or a bar chart in a business program.

In short, the information bar is handy for displaying anything you want to keep separate from the content region. Of course, you pay a price for this convenience in that the information bar takes up space that would otherwise be part of the content region.

Active and Inactive Windows

Although you can display several windows on the screen at the same time, only one of them may be *active*; the rest are *inactive*. The active window is the only one the user can operate on — that is, move, scroll, draw in, or whatever. To activate an inactive window, simply click the mouse anywhere in it. This moves the window to the front and displays any controls it has. As you may recall from Chapter 8, it also generates an activate event, the event that has the highest priority.

Fundamental Tool Calls for Windows

To display a window, your program must do these things:

1. Start QuickDraw and the Event Manager — the Window Manager needs the services of both of them.
2. Start the Window Manager with a `WindStartup` call.
3. Define the window with a `NewWindow` call. `NewWindow` creates a `GrafPort` for the window and stores the entire `GrafPort` record plus information about the window's characteristics (e.g., its size and position and which controls it has) in a large *window record* in memory.
4. Activate the window's `GrafPort` with a `SetPort` call to QuickDraw.
5. Draw the image the window is to display, using Quickdraw calls.
6. Give a `ShowWindow` call to display the completed window.

Table 9-1 summarizes these Memory Manager tool calls, along with some others that application programs need to work with windows. The *Housekeeping Calls* include `WindStartup` and `WindShutDown`, plus calls that let you create and display a new window and close an existing one. Of these six tool calls, `NewWindow` is the most complex.

NewWindow

The `NewWindow` call requires you to supply a pointer to a parameter table for the window you want to create, and returns a pointer to the `GrafPort` it sets up for that window. Table 9-2 lists the entries in `NewWindow`'s parameter table. The table is quite long (24 entries in all), but don't be intimidated by it. Unless you're creating a window that uses all the controls, many of the entries will be zero. Let's take the entries in the order they appear in the table.

The first parameter, `param__length`, gives the total length of the table (including this entry) in bytes. `NewWindow` checks `param__length` against the byte count it expects and reports an error if they don't match. This ensures that you have supplied the correct number of entries. The easiest way to get the `paramlength` value is to label the beginning and end of the table, and let the assembler calculate the length, as in this example:

```

WinlParamTable anop
    dc i'WinlEnd-WinlParamTable' ;Length of table
    . .
    . . (23 more parameter table entries)
    . .
WinlEnd anop

```

Table 9-1

Housekeeping Calls

__WindStartup	Start the Window Manager
Call with: PushWord <i>ProgramID</i>	;The program's ID number
__WindStartup	
Result: None	
Note:	The Window Manager shares the Event Manager's space in bank 0.
__Refresh	Redraw the screen
Call with: PushLong #0	
__Refresh	
Result: None	
Note:	Clears the screen and redraws it. Generally, you should call Refresh immediately after you do a WindStartup.
__NewWindow	Create a new window using specified parameters
Call with: PushLong #0	;Space for result
PushLong <i>ParamPtr</i>	;Pointer to parameter table
__NewWindow	
Result: Pointer to new window's GrafPort (long word)	
Note:	See text for description of the window parameter table.
__ShowWindow	Make the specified window visible
Call with: PushLong <i>WindPortPtr</i>	;Pointer to window's port
__ShowWindow	
Result: None	
Note:	Makes the window visible and draws it. Does not change the front-to-back ordering of the windows; see also SelectWindow.
__CloseWindow	Close the specified window
Call with: PushLong <i>WindPortPtr</i>	;Pointer to window's port
__CloseWindow	
Result: None	
Note:	Removes the window from the screen and destroys its window record.
__WindShutDown	Shut down the Window Manager
Call with: __WindShutDown	
Result: None	

Table 9-1 (cont.)

Window-Shuffling Calls	
<code>__HideWindow</code>	Make the specified window invisible
Call with: <code>PushLong WindPortPtr</code> ;Pointer to window's port	
<code>__HideWindow</code>	
Result: None	
Note: If this is the frontmost (active) window, <code>HideWindow</code> makes the window behind it active and generates the appropriate activate events.	
<code>__SelectWindow</code>	Make the specified window active
Call with: <code>PushLong WindPortPtr</code> ;Pointer to window's port	
<code>__SelectWindow</code>	
Result: None	
Note: Unhighlights the currently active window, brings the specified window to the front, highlights it, and generates the appropriate activate events. Should be called if there's a mouse-down event in the content region of an inactive window.	

The *wFrame* parameter is a word-size bit pattern that is arranged as shown in Figure 9-5. Here, Title Bar, Close Box, Right Scroll Bar, and so on, are switches by which you specify the controls the window should have; a 1 means the window has that control, a 0 means it does not. A 1 in the "Movable" bit specifies that the window may be dragged by its title bar. The "Visible" bit stipulates whether the window should be visible (1) or invisible (0). Generally, you should make it invisible to start. Then, after you have drawn its contents, do a `ShowWindow` call to make it visible.

Some of these bits are interrelated, and you should observe the following rules when constructing a *wFrame* word:

- Since the close box and zoom box are part of the title bar, if the window is to have either or both boxes, it must also have a title bar. In other words, if bit 8 or 14 is set to 1, bit 15 must also be set to 1.
- A window must also have a title bar if it is to be movable (bit 7 = 1).
- To have a grow box, a window must also have a scroll bar. Thus, to set bit 10 to 1, bit 11 or 12 must also be 1.

Because each bit is significant, you should set up *wFrame* with a DC directive whose operand is a 16-bit binary (%) pattern. Some examples are:

```

wFrame dc i '%0000000000000000'      ;No controls
wFrame dc i '%1000000010000000'      ;Title bar
                                      ; (can be dragged)
wFrame dc i '%1100000110000000'      ;Title bar, close
                                      ; and zoom boxes
wFrame dc i '%1101110110100000'      ;All controls and
                                      ; information bar

```

Using a binary pattern not only helps make your parameter table more understandable, but it makes troubleshooting easier if a particular control (say, the zoom box) is missing when the window is displayed.

The next item in the parameter table, *wTitle*, is a pointer to the text for the window's title (or 0, if the window has no title bar). Set up *wTitle* with this kind of directive:

Table 9-2

Name	Data Type	Description
param__length	Word	Number of bytes in parameter table.
wFrame	Word	Bit vector that describes the window.
wTitle	Long word	Pointer to window's title.
wRefCon	Long word	Reserved for application's use.
wZoom	Rect	Position of content when zoomed.
wColor	Long word	Pointer to window's color table.
wYOrigin	Word	Vertical offset of content region.
wXOrigin	Word	Horizontal offset of content region.
wDataH	Word	Height of data area (rows).
wDataW	Word	Width of data area (columns).
wMaxH	Word	Maximum height to which content region can grow.
wMaxW	Word	Maximum width to which content region can grow.
wScrollVer	Word	Number of pixels to scroll content vertically.
wScrollHor	Word	Number of pixels to scroll content horizontally.
wPageVer	Word	Number of pixels to page content vertically.
wPageHor	Word	Number of pixels to page content horizontally.
wInfoRefCon	Long word	Value passed to information bar draw routine.
wInfoHeight	Word	Height of information bar (rows).
wFrameDefProc	Long word	Pointer to routine that draws the window's shape.
wInfoDefProc	Long word	Pointer to routine that draws the information bar's interior.
wContDefProc	Long word	Pointer to routine that draws the content region's interior.
wPosition	Rect	Content region's screen coordinates.
wPlane	Long word	Window's order.
wStorage	Long word	Address of memory to use for window record.

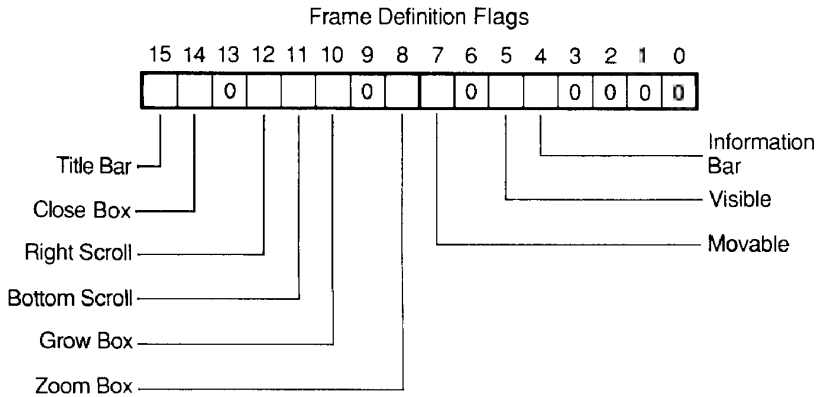


Figure 9-5

```
wTitle dc i4'WinlTitle'
```

where `WinlTitle` points to a statement such as:

```
WinlTitle str 'Window 1'
```

`wRefCon` is an application-defined reference value. In general, you should enter `dc i4'0'`.

`wZoom` defines the screen coordinates of the content region when the window is zoomed. This takes a directive of the form:

```
dc i'V1,H1,V2,H2'
```

Generally, you will want the zoomed window to fill the entire screen, so if it has no title bar or scroll bars, you would enter (in 640 mode):

```
dc i'0,0,199,639'
```

If your window has a title bar or scroll bar, however, you must account for the space they occupy. The title bar is 26 rows high (rows 0 through 25), the bottom scroll bar is nine rows high (191 through 199), and the right scroll bar is 19 columns wide in 640 mode (621 through 639). Hence, the `wZoom` rectangle for a 640 mode window that has only a title bar is defined by:

```
dc i'26,0,199,639'
```


A window with a title bar and both scroll bars is defined by:

```
dc i '26,0,190,620'
```

wColor points to the color table used to draw the window's frame. Entering 0 calls up the default color table.

wYOrigin and *wXOrigin* are the number of rows and columns the content region is offset from the top left-hand corner of the data area. If you set both values to 0 (the usual case), the window's content region will show the beginning of the data area.

wDataH and *wDataW* are the height and width of the data area; that is, the entire picture.

wMaxH and *wMaxW* are the maximum height and width to which the window can grow. To let the window fill the screen, enter 200 for *wMaxH* and either 320 or 640 for *wMaxW*. If the window has no grow box, enter 0 for both parameters.

wScrollVer is the number of pixels the content region should scroll vertically when the user clicks in one of the right scroll bar's arrows. Make this a small number, such as 4. Similarly, *wScrollHor* is the number of pixels the content region should scroll horizontally when the user clicks in one of the bottom scroll bar's arrows. Because the screen is wider than it is tall, make *wScrollHor* larger than *wScrollVer*; a value of 8 would be appropriate. Of course, if your window has no scroll bars, set both parameters to 0.

wPageVer and *wPageHor* are the number of pixels the content region should scroll (or "page") when the user clicks in a scroll bar's gray area. These values should be much larger than *wScrollVer* and *wScrollHor*. It is appropriate to make them, say, ten times larger, in which case *wPageVer* and *wPageHor* would be 40 and 80, respectively. As before, if your window has no scroll bars, set both parameters to 0.

wInfoRefCon is a long-word value passed to the routine that fills in the information bar. It may be, perhaps, a pointer to a string the routine should display. *wInfoHeight* is the height of the information bar. Make both parameters 0 if your window has no information bar.

The next three parameters hold pointers to various drawing routines. *wFrameDefProc* points to a routine that defines the window's shape. You would only use it if you want the window to have a shape other than the standard rectangle; in general, set *wFrameDefProc* to 0. *wInfoDefProc* points to a routine that fills in the information bar.

wContDefProc points to a routine that draws the window's content region. The routine must end with an RTL (Return from Subroutine Long)

instruction because it is called by the Window Manager, which may reside in a different bank than your program.

wPosition holds the initial screen coordinates of the window's content region, which specify its position and size. These coordinates become the PortRect of the window's GrafPort.

wPlane indicates where this window is to appear in a stack of overlapping windows. Usually you should create the windows in back-to-front order, and set *wPlane* to -1 in each window's parameter table. But you needn't create all the windows at the same time. You can create a new window at any time by issuing a NewWindow call. If the new window is to appear atop the others, set its *wPlane* value to -1 ; if it is to appear beneath the others (at the bottom of the stack), set *wPlane* to zero.

The final parameter in the table, *wStorage*, points to the memory location where the window's record should be stored. You can allocate this space yourself, but because Apple may change the length of the window record sometime in the future, it's safer to let the Window Manager allocate the space. To hand this job over to the Window Manager, set *wStorage* to zero.

The window record that NewWindow creates is 325 bytes long, the longest record in an Apple IIGS program. Fortunately, you never work with it directly; rather, you access it indirectly, through calls to the Window Manager.

You also access the window record indirectly when you make QuickDraw tool calls, because the window's GrafPort record is part of its window record. In fact, NewWindow returns a pointer to this GrafPort, or *Window Manager port*, on the stack. As Table 9-1 shows, this pointer is required input for SelectWindow, HideWindow, and other window calls.

In short, then, NewWindow creates a window record in memory that contains a copy of the current GrafPort record, makes this window (and its Window Manager port) active, and returns a pointer to its port. Because the Window Manager port is active, any calls you make to QuickDraw will only apply to this port.

Having an individual port for each window is convenient in that any changes you make with QuickDraw (such as moving the pen or increasing its size) affect only the active window. Every other window retains its own drawing environment. Thus, you never have to worry about the current pen attributes, color table, or anything else when you switch windows; the window you activate will operate under its own port settings.

ShowWindow

The ShowWindow call displays the window NewWindow has created. Your program should do a ShowWindow after it has drawn a window's content region.

CloseWindow

The `CloseWindow` call deletes a window entirely by removing it from the screen and destroying its window record. If you simply want to hide a window, and keep it available for displaying later, you can make it invisible by calling `HideWindow`.

Window-Shuffling Calls

The `HideWindow` call makes a window invisible, but (unlike `CloseWindow`) keeps its record in memory. If the window is at the top of a stack of windows, `HideWindow` also makes the window beneath it active. Generally, your program should call `HideWindow` if the user clicks the mouse in the active window's close box.

The `SelectWindow` call makes a window active by bringing it to the front and highlighting it. You would normally do a `SelectWindow` if the user has pressed the mouse button in the content region of an inactive window. `SelectWindow` is also useful for revealing a window that was hidden previously. Here, the program should allow the user to unhide a window by selecting its name from a menu.

At this point, you may be expecting me to present a program that puts some overlapping windows on the screen and lets you shuffle them around. I have already described the tool calls necessary to do that, so it's reasonable to anticipate such an example. However, I will forgo the programming for the moment because it would entail using the Event Manager's `GetNextEvent` call. `GetNextEvent` would require you to do a lot of tasks manually that are performed automatically by a Window Manager resource called the `TaskMaster`.

The TaskMaster

The Window Manager's `TaskMaster` tool is an enhanced version of the Event Manager's `GetNextEvent` tool. In fact, `TaskMaster` actually calls `GetNextEvent` internally. Therefore, if the Window Manager is active (as it will be for most applications), you can *forget about* `GetNextEvent` entirely and use `TaskMaster` instead!

Like `GetNextEvent`, `TaskMaster` checks the Event Manager's event list and returns a 0 (the null event) if no event is pending. If an event *is* pending, however, `TaskMaster` does not simply pass it to your program, as `GetNextEvent` does, but tries to handle the event itself. For example, if the user

presses the mouse button in the active window's zoom box, TaskMaster does everything necessary to zoom the window to its predefined full size. Once it has zoomed the window, TaskMaster returns a null event to your program, as if the zoom operation had never occurred!

Calling TaskMaster

Here is a summary of the TaskMaster tool call:

```

_TaskMaster          Get the next event using TaskMaster
Call with:  PushWord #0                ;Space for result
            PushWord EventMask       ;Event mask for
                                           ; GetNextEvent
            PushLong TaskRecPtr      ;Pointer to extended
                                           ; event record
            _TaskMaster
Result:  Task code (word)

```

Note that like the Event Manager's GetNextEvent call, the Window Manager's TaskMaster call requires two inputs: an event mask and a pointer to an event record. The event mask is the same as the one used by GetNextEvent (see Figure 8-2), but TaskMaster's *task record* has two more entries than GetNextEvent's event record. Specifically, a task record contains these seven entries:

What	word	Event record entries, same as
Message	long word	for GetNextEvent
When	long word	
Where	point	
Modifiers	word	
TaskData	long word	Additional entries for TaskMaster
TaskMask	long word	

Here, TaskData is a long word in which TaskMaster returns information pertinent to a window event. This is usually a pointer to the window's port. In your programs, define TaskData with a DS 4 directive.

The TaskMask (shown in Figure 9-6) is a 32-bit pattern whose 13 low bits indicate which operations you want *TaskMaster*, rather than your program, to perform. You can, for example, tell TaskMaster to do everything necessary when the user scrolls, drags, closes, zooms, or grows a window. You can also tell TaskMaster to redraw the active window and make it inactive

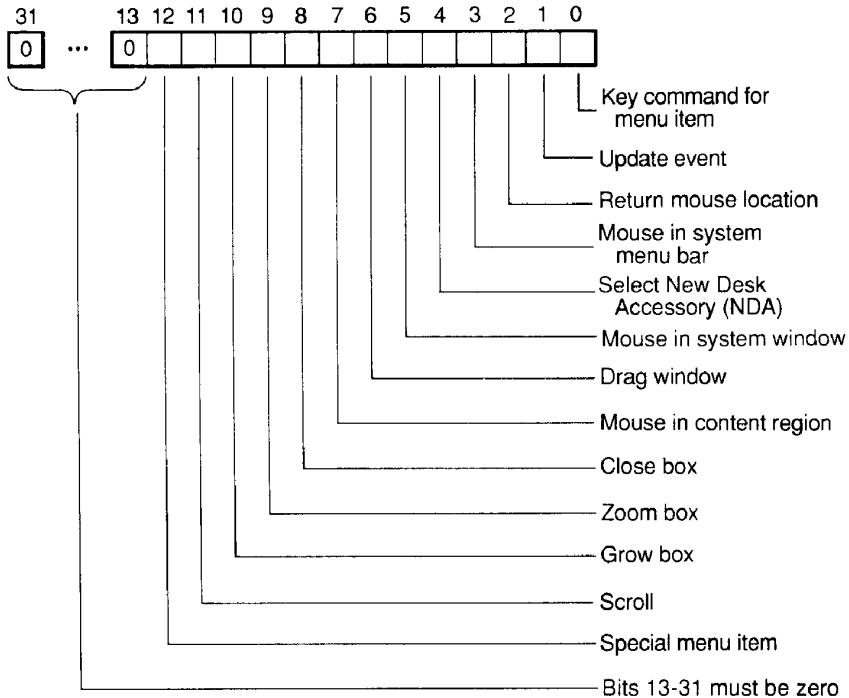


Figure 9-6

(i.e., *update* it) when the user clicks in an inactive window. To hand these tasks to TaskMaster, set all 13 low bits in the TaskMask to 1 by defining it with a DC `i4'$01FFF'` directive.

Although TaskMaster relieves your program of a lot of standard operations, the program must still deal with events that TaskMaster doesn't know how to handle. These are key-down, auto-key, activate, switch, desk accessory, and the user-defined events. If your program is to accept any of these events, it must include subroutines that process them.

Processing Events

TaskMaster returns a word-size *task code* (a number from 0 to 28) on the stack. If no event is pending, or if TaskMaster has already processed an event (as directed by your TaskMask), the task code is 0 — the number of our old friend, the null event. Otherwise, TaskMaster returns one of the codes listed in Table 9-3.

Table 9-3

Event Manager events

- 0 – Null event
- 1 – Mouse-down
- 2 – Mouse-up
- 3 – Key-down
- 4 – Undefined
- 5 – Auto-key
- 6 – Update
- 7 – Undefined
- 8 – Activate
- 9 – Switch
- 10 – Desk accessory
- 11 – Device driver
- 12 – Application-defined
- 13 – Application-defined
- 14 – Application-defined
- 15 – Application-defined

TaskMaster events

- | | | | |
|------|----------------------|----|------------------|
| 16 – | Mouse button pressed | in | desktop (screen) |
| 17 – | " | " | " |
| 18 – | " | " | " |
| 19 – | " | " | " |
| 20 – | " | " | " |
| 21 – | " | " | " |
| 22 – | " | " | " |
| 23 – | " | " | " |
| 24 – | " | " | " |
| 25 – | " | " | " |
| 26 – | " | " | " |
| 27 – | " | " | " |
| 28 – | " | " | " |

The first 16 codes are the event codes that can result from TaskMaster making a `GetNextEvent` call to the Event Manager. Of these events, TaskMaster can process only `Update` (code 6) on its own. It passes all other codes to your application program, because, as after all, it has no way of knowing what you want these events to do.

The list of events your program must handle is normally quite short, however. For starters, you will probably never receive a mouse-down or mouse-up code, because when TaskMaster receives one, it says, "My boss, the Window Manager, will want to know about this," and translates the code

into one of the “mouse button pressed” codes at the bottom of Table 9-3. Furthermore, if you are not providing for any switch, desk accessory, device driver, or application-defined events, your program needn’t deal with codes between 9 and 15. That leaves only four event types to contend with: null, key-down, auto-key, and activate. If the program is entirely mouse-driven, and ignores key presses, the list is even shorter.

If your program has defined the TaskMaster task mask input with DC i4’\$01FFF’(as I suggested earlier), it must process only three task codes in the bottom portion of the list: system menu bar (17), content region (19), and close box (22). Menus will be discussed later, so you can disregard that topic for now. When the user presses the mouse button in the content region of the active window, TaskMaster stores a pointer to the window in the TaskData field of the task record and returns task code 19 on the stack. TaskMaster can’t process this event itself because the content region contains a picture the application has drawn, and the mouse may signify something in some applications.

When the user presses the mouse button in a window’s close box (or *go-away region*, in Apple’s documentation), TaskMaster stores a pointer to the window in the TaskData field of the task record and returns task code 22 on the stack. In general, your program should respond by calling `HideWindow` to make the window disappear. (How can the user make the window reappear? He or she normally does that by selecting from a menu.)

Tool Sets Required by TaskMaster

In the course of its work, TaskMaster checks for events from the Control Manager and the Menu Manager, so both of these tool sets must be active. I will discuss these managers in detail later, but for now I will simply present the start-up and shut-down calls that must be included in every program that calls TaskMaster (see Table 9-4).

These tool sets are RAM-based, so you must read them into memory by calling `LoadTools`. Furthermore, as you can see from the start-up calls, each manager requires one page in memory, so you must account for that when you call `NewHandle`.

An Example Window Program

I’ve covered a lot of window theory in this chapter, and you’re probably eager to try some of it. Clearly, it’s time to do some programming!

Table 9-4

<u>__CtlStartup</u>		Start the Control Manager
Call with:	PushWord <i>ProgramID</i> ;Program ID	
	PushWord <i>DirectPageLoc</i> ;Loc. of 1-page work area	
	<u>__CtlStartup</u>	
Result:	None	
<u>__CtlShutDown</u>		Shut down the Control Manager
Call with:	<u>__CtlShutDown</u>	
Result:	None	
<u>__MenuStartup</u>		Start the Menu Manager
Call with:	PushWord <i>ProgramID</i> ;Program ID	
	PushWord <i>DirectPageLoc</i> ;Loc. of 1-page work area	
	<u>__MenuStartup</u>	
Result:	None	
<u>__MenuShutDown</u>		Shut down the Menu Manager
Call with:	<u>__MenuShutDown</u>	
Result:	None	

The program listed in Example 9-1 (WINDOWS) displays three overlapping windows on a 640 mode screen, as shown in Figure 9-7. Each has a title bar, zoom and grow boxes, and both scroll bars — and all controls are operable. That is, you can drag a window by its title bar, make it zoom or grow, and scroll through its content region. WINDOWS keeps its windows on the screen until you press Esc.

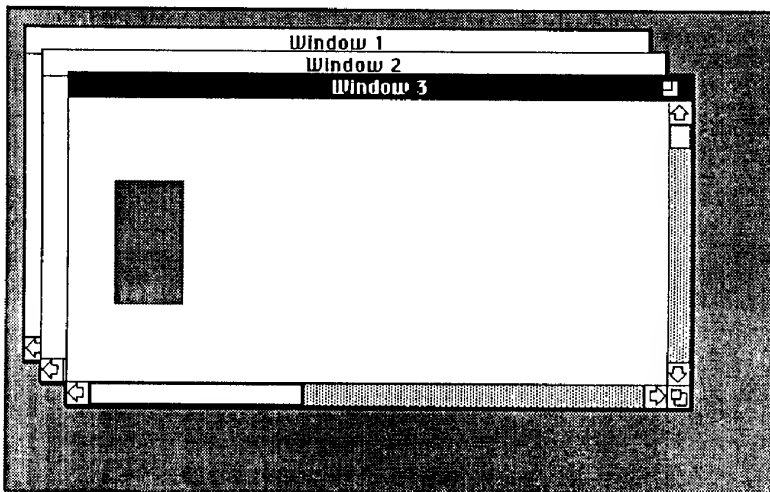


Figure 9-7

**Example 9-1**

; WINDOWS displays three windows and lets the user shuffle or
; drag them, or make them zoom or grow. To quit, press Esc.

```

      absaddr on
      MCOPY windows.macros

Windows      START
              using GlobalData

;-----
;
; Global equates used throughout the program.

ScreenMode   gequ $80                ;640 mode, no fill
MaxX          gequ 640                ;640 mode for Event Manager

              phk                      ;Set data bank to program
              plb                      ; bank to allow absolute addressing

              jsr InitStuff            ;Initialize everything
              bcs AllDone              ;Quit if initialization fails

              stz QuitFlag             ;Initialize the quit flag to 0
              jsr EventLoop           ;Let the user play

AllDone       anop                    ;All is done, shut down
              PushLong Win1Ptr         ;Close the windows
              _CloseWindow
              PushLong Win2Ptr
              _CloseWindow
              PushLong Win3Ptr
              _CloseWindow

              _DeskShutDown            ;Desk Manager
              _MenuShutDown           ;Menu Manager
              _WindShutDown           ;Window Manager
              _CtlShutDown            ;Control Manager
              _EMShutDown             ;Event Manager
              _QDShutDown             ;QuickDraw II
              _MTShutDown             ;Miscellaneous Tools

              PushWord MyID            ;Discard the program's handle
              _DisposeAll

              PushWord MyID
              _MMShutDown             ;Memory Manager
              _TLShutDown            ;Tool Locator

              _Quit QuitParams         ;Do a ProDOS Quit call
              brk $F0                 ;If it fails, break

      END

;*****
;
; Global Data
;
;*****

```

314 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

```

GlobalData    DATA

QuitParams    dc i'0'                ;Return to caller
               dc i'$4000'           ;Make program restartable in memory

ToolTable     dc i'NumTools'         ;No. of tool sets in table
               dc i'4,$0100'         ;QuickDraw
               dc i'5,$0100'         ;Desk Manager
               dc i'6,$0100'         ;Event Manager
               dc i'14,$0100'        ;Window Manager
               dc i'15,$0100'        ;Menu Manager
               dc i'16,$0100'        ;Control Manager
TTEnd         anop

TableSize     equ TTEnd-ToolTable-2
NumTools      equ TableSize/4

TaskRecord    anop                  ;Buffer for task record
EventWhat     ds 2
EventMessage  ds 4
EventWhen     ds 4
EventWhere    ds 4
EventModifiers ds 2
TaskData      ds 4
TaskMask      dc i'$01FFF'          ;Make TaskMaster do all it can

Win1Ptr       ds 4                  ;Window pointers
Win2Ptr       ds 4
Win3Ptr       ds 4

MyID          ds 2                  ;This will hold the program's i.d.

VolNotFound   equ $45               ;ProDOS error

QuitFlag      ds 2                  ;Quit flag
               END

;*****
;
; InitStuff
;
; Initializes tool sets, gets space in bank 0 for use as zero
; page by tool sets that need it, and ensures that RAM-based
; tools are in memory.
;
;*****

InitStuff     START
               using GlobalData

; Initialize the Tool Locator, Memory Manager, and Miscellaneous
; Tools.

               _TLStartup            ;Tool locator

               PushWord #0           ;Memory Manager
               _MMStartup

               pla                    ;Memory Manager returns program's ID
               sta MyID

```

```

        _MTStartup                ;Misc. Tools
        ldx #3
        jsr PrepareToDie

; Get some space for the direct page we need. QuickDraw needs
; three pages and each Manager except Window needs one page.

        PushLong #0                ;Space for handle
        PushLong #$600            ;Six pages
        PushWord MyID             ;Owner
        PushWord #$C005           ;Locked, fixed, fixed bank
        PushLong #0              ;Location
        _NewHandle
        ldx #$FF
        jsr PrepareToDie

        pla                      ;Read handle and store in dp
        sta 0
        pla
        sta 2

        lda [0]                  ;Get dp location from handle
        sta 4                    ; and store at loc 4 of dp

; Initialize QuickDraw

        pha                      ;Use dp obtained from handle
        PushWord #ScreenMode      ;Mode = 640
        PushWord #160            ;Max size of scan line (in bytes)
        PushWord MyID
        _QDStartup
        ldx #4
        jsr PrepareToDie

; Initialize Event Manager

        lda 4                    ;dp to use = QD dp + $300
        clc
        adc #$300
        sta 4
        pha
        PushWord #20             ;Queue size
        PushWord #0              ;X clamp left
        PushWord #MaxX           ;X clamp right
        PushWord #0              ;Y clamp top
        PushWord #200            ;Y clamp bottom
        PushWord MyID
        _EMStartup
        ldx #6
        jsr PrepareToDie

;-----
;
; Load the RAM-based tools I need.

LoadAgain    PushLong #ToolTable
              _LoadTools
              bcc ToolsLoaded
              cmp #VolNotFound
              beq DoMount
              sec

```

316 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

```

        ldx #$FE
        jsr PrepareToDie

DoMount      anop
              jsr MountBootDisk
              cmp #1
              beq LoadAgain

              sec
              rts

; The tools have been loaded. Initialize the Window Manager, Control
; Manager, Menu Manager, and Desk Manager.

ToolsLoaded  anop
              PushWord MyID           ;Window Manager
              _WindStartup
              ldx #14
              jsr PrepareToDie

              PushLong #0             ;Prepare screen for windows
              _RefreshDesktop

              PushWord MyID           ;Control Manager
              lda 4                   ;dp to use = EM dp + $100
              clc
              adc #$100
              sta 4
              pha
              _CtlStartup
              ldx #16
              jsr PrepareToDie

              PushWord MyID           ;Menu Manager
              lda 4
              clc
              adc #$100
              pha
              _MenuStartup
              ldx #15
              jsr PrepareToDie

              _DeskStartup            ;Desk Manager

              jsr SetUpWindows        ;Show the windows
              _ShowCursor             ; and the mouse pointer

              jsr DrawWindows         ;Draw the window contents

              clc                     ;Clear the carry flag
              rts                     ; and return

END

; *****
;
; Set Up Windows
;
; Opens the three windows (but does not draw their contents).
;
; *****

```

```

SetUpWindows  START
               using GlobalData
               using WindowData

               PushLong #0                ;Window 1
               PushLong #Win1ParamBlock
               _NewWindow

               pla
               sta Win1Ptr
               pla
               sta Win1Ptr+2

               PushLong #0                ;Window 2
               PushLong #Win2ParamBlock
               _NewWindow

               pla
               sta Win2Ptr
               pla
               sta Win2Ptr+2

               PushLong #0                ;Window 3
               PushLong #Win3ParamBlock
               _NewWindow

               pla
               sta Win3Ptr
               pla
               sta Win3Ptr+2

               rts
               END

;*****
;
; Event Loop
;
; Display windows until user presses Esc.
;
;*****

EventLoop     START
               using GlobalData

Again         lda QuitFlag
               bne NoMore

               PushWord #0                ;Space for result
               PushWord #$FFFF            ;Accept any event
               PushLong #TaskRecord       ;Point to task record buffer
               _TaskMaster

               pla                        ;Is an event available?

               beq Again                  ; No. Continue polling
               asl a                      ; Yes. Double it
               tax                        ; and copy it into X.

               jsr (TaskTable,x)          ;Execute the event's routine,
               bra Again                  ; then resume polling

NoMore        rts

```

318 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

```

TaskTable      anop                      ;Event Manager events
                dc i'Ignore'             ; 0 null
                dc i'Ignore'             ; 1 mouse down
                dc i'Ignore'             ; 2 mouse up
                dc i'DoKeyDown'           ; 3 key down
                dc i'Ignore'             ; 4 undefined
                dc i'Ignore'             ; 5 auto-key
                dc i'Ignore'             ; 6 update
                dc i'Ignore'             ; 7 undefined
                dc i'Ignore'             ; 8 activate
                dc i'Ignore'             ; 9 switch
                dc i'Ignore'             ; 10 desk acc
                dc i'Ignore'             ; 11 device driver
                dc i'Ignore'             ; 12 ap
                dc i'Ignore'             ; 13 ap
                dc i'Ignore'             ; 14 ap
                dc i'Ignore'             ; 15 ap
                dc i'Ignore'             ; 16 in desktop
                dc i'Ignore'             ; 17 in system menu bar
                dc i'Ignore'             ; 18 in system window
                dc i'Ignore'             ; 19 in window content region
                dc i'Ignore'             ; 20 in drag region (title bar)
                dc i'Ignore'             ; 21 in grow box
                dc i'Ignore'             ; 22 in go-away region (close box)
                dc i'Ignore'             ; 23 in zoom box
                dc i'Ignore'             ; 24 in information bar
                dc i'Ignore'             ; 25 in right scroll bar
                dc i'Ignore'             ; 26 in bottom scroll bar
                dc i'Ignore'             ; 27 in frame
                dc i'Ignore'             ; 28 in drop region
                END

```

```

;*****
;
; Ignore
;
; This do-nothing subroutine returns to the event loop for
; events we don't want.
;
;*****

```

```

Ignore      START
            rts
            END

```

```

;*****
;
; DoKeyDown
;
; If user pressed Esc, DoKeyDown sets the QuitFlag to 1. It
; ignores any other key.
;
;*****

```

```

DoKeyDown    START
            using GlobalData

            lda EventMessage          ;Read the ASCII character
            cmp #27                   ;Is it Esc?
            bne Exit

```



```

        dc      i2'160'          Number of pixels to page horizontally.
        dc      i4'0'            Infomation bar text string.
        dc      i2'0'            Info bar height
        dc      i4'0'            Routine to draw shape (none, standard)
        dc      i4'0'            Routine to draw info. bar.
        dc      i4'DrawWin2'     Routine to draw content.
        dc      i'50,30,110,380' Size/pos of content
        dc      i4'-1'          Window's order (-1 means topmost)
        dc      i4'0'          Window Manager allocates wind. record
Win2End  anop

Win3ParamBlock anop
        dc      i'Win3End-Win3ParamBlock'
        dc      i2'$1001110110000000' Everything but close box & info bar
        dc      i4'Win3Title'     Pointer to window's title
        dc      i4'0'            Reserved (wRefCon)
        dc      i2'26,0,190,620' Zoomed content region
        dc      i4'0'            Default color table
        dc      i2'0'            Vertical origin
        dc      i2'0'            Horizontal origin
        dc      i2'200'          Data area height
        dc      i2'640'          Data area width
        dc      i2'200'          Max grow height
        dc      i2'640'          Max grow width
        dc      i2'4'            Number of pixels to scroll vertically.
        dc      i2'16'          Number of pixels to scroll
horizontally.
        dc      i2'40'          Number of pixels to page vertically.
        dc      i2'160'         Number of pixels to page horizontally.
        dc      i4'0'            Infomation bar text string.
        dc      i2'0'            Info bar height
        dc      i4'0'            Routine to draw shape (none, standard)
        dc      i4'0'            Routine to draw info. bar.
        dc      i4'DrawWin3'     Routine to draw content.
        dc      i'60,40,120,400' Size/pos of content
        dc      i4'-1'          Window's order (-1 means topmost)
        dc      i4'0'          Window Manager allocates wind. record
Win3End  anop

Win1Title      str 'Window 1'
Win2Title      str 'Window 2'
Win3Title      str 'Window 3'

```

```

PenPat1  dc h'11 11 11 11 11 11 11 11' ;Dithered blue pen pattern
        dc h'11 11 11 11 11 11 11 11'
        dc h'11 11 11 11 11 11 11 11'
        dc h'11 11 11 11 11 11 11 11'

Rect      dc i'20,20,40,40'          ;Rectangle painted in windows

```

END

```

;*****
;
; Draw Windows
;
; Fills in the content region of each window, working rearward.
;
;*****

```



```
DrawWindows    START
    using GlobalData
```

```
    PushLong Win3Ptr          ;Window 3
    _SetPort
```

```
    jsl DrawWin3
```

```
    PushLong Win3Ptr
    _ShowWindow
```

```
    PushLong Win2Ptr          ;Window 2
    _SetPort
```

```
    jsl DrawWin2
```

```
    PushLong Win2Ptr
    _ShowWindow
```

```
    PushLong Win1Ptr          ;Window 1
    SetPort
```

```
    jsl DrawWin1
```

```
    PushLong Win1Ptr
    _ShowWindow
```

```
    rts
    END
```

```
;*****
;
; Draw Window 1
;
; Draws the contents of Window 1--a blue box on a white
; background. Called by DrawWindows initially and by the Window
; Manager when it updates the window.
;
;*****
```

```
DrawWin1 START
    using WindowData
```

```
    PushWord #3                ;Background is white
    _SetSolidBackPat
```

```
    PushLong #PenPat1          ;Set pen pattern to blue
    _SetPenPat
```

```
    PushLong #Rect             ; and paint the rectangle
    _PaintRect
```

```
    rtl                        ;RTL required for update events
    END
```

322 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

```
;*****
;
; Draw Window 2
;
; Draws the contents of Window 2--a white box on a green
; background. Called by DrawWindows initially and by the Window
; Manager when it updates the window.
;
;*****

DrawWin2 START
    using WindowData

    PushWord #2                ;Background is green
    _SetSolidBackPat

    PushWord #3                ;Set pen pattern to white
    _SetSolidPenPat

    PushLong #Rect             ; and paint the rectangle
    _PaintRect

    rtl                        ;RTL required for update events
    END

;*****
;
; Draw Window 3
;
; Draws the contents of Window 3--a green box on a red
; background. Called by DrawWindows initially and by the Window
; Manager when it updates the window.
;
;*****

DrawWin3 START
    using WindowData

    PushWord #1
    _SetSolidBackPat

    PushWord #2                ;Set pen pattern to green
    _SetSolidPenPat

    PushLong #Rect             ; and paint the rectangle
    _PaintRect

    rtl                        ;RTL required for update events
    END

*****
*
* Prepare To Die
*
* Dies if carry is set.
*
* Error number to display is in X register.
* Assumes that Miscellaneous Tools is active.
*
*****
```

```

PrepareToDie  START
               bcs RealDeath      ;Carry = 1?
               rts                 ; No. Return to caller

RealDeath     phx                   ; Yes. Goodbye, program.
               PushLong #DeathMsg
               _SysFailMgr

DeathMsg      str 'Could not handle error '

               END

;*****
;
; Mount Boot Disk
;
; Tells the user to insert the disk that contains the RAM-based
; tools.
;
;*****

MountBootDisk START

               _Set_Prefix SetPrefixParams
               _Get_Prefix GetPrefixParams

               PushWord #0           ;Space for result
               PushWord #195         ;Column position for dialog box
               PushWord #30          ;Row position for dialog box
               PushLong #PromptStr   ;Prompt at top of dialog box
               PushLong #VolStr      ;Volume name string
               PushLong #OKStr       ;String in Button 1
               PushLong #CancelStr   ;String in Button 2
               _TLMountVolume

               pla                   ;Obtain the button number
               rts                   ; and return to caller

PromptStr     str 'Please insert the disk.'
VolStr        ds 16
OKStr         str 'OK'
CancelStr     str 'Shutdown'

GetPrefixParams dc i'7'
               dc i4'VolStr'

SetPrefixParams dc i'7'
               dc i4'BootStr'

BootStr       str '*/'

               END

```

I *could* have displayed a complex pixel image in each window, but that would have made the program longer and more difficult to understand. Instead, I used QuickDraw to draw a colored box in each one. Window 1

has a blue box on a white background, Window 2 has a white box on a green background, and Window 3 has a green box on a red background. This may not be very creative on my part, but it serves to keep the program to minimal length. Once you get WINDOWS up and running, you may want to modify it by inserting QuickDraw calls of your own.

The WINDOWS program is quite long, but if you understand the earlier programs in this book and the material in this chapter, you shouldn't have any problem with it. Here is how WINDOWS operates.

To begin, the Windows segment calls `InitStuff` to initialize the tools and set up the windows. `InitStuff` starts the same tools as the earlier `SHOWTEXT` program (Example 8-3), but it also starts the Window Manager, the Menu Manager, and the Control Manager. (The Menu and Control Managers must be active for TaskMaster to operate.) When the tools have been initialized, `InitStuff` calls `SetUpWindows` to display the windows.

`SetUpWindows` performs three `NewWindow` calls, one for each window, based on parameter tables in the `WindowData` segment. By examining the `wFrame` parameter (the second entry in each table), you will notice that I have made each window invisible and have given it a movable title bar, grow and zoom boxes, and scroll bars. I have not provided an information bar because it is unnecessary. I haven't provided a close box either, because I have not yet described how to reinstate a window once it has been hidden. (That's coming in Chapter 10.)

Each window parameter table also includes a pointer to a subroutine (`DrawWin1`, `2`, or `3`) that the Window Manager will use to draw the contents of an inactive window when you activate it — that is, when you press the mouse button anywhere in it. Each `DrawWin` subroutine sets the background and pen patterns (i.e., their colors), then paints a rectangle.

`DrawWin2` and `DrawWin3` use `SetSolidPenPat` to set the color, because they draw with colors from the first four entries of the 640 mode color table. `DrawWin1` must use `SetPenPat`, because it draws with blue, which can only be obtained through dithering. (If you specify color 5, blue in the 640 mode table, Quickdraw will ignore the 1 in bit 3 and draw the rectangle using color 1, red.)

Note that each `DrawWin` subroutine ends with an *rtl* (Return from Subroutine Long) rather than an *rts* (Return from Subroutine). They must be "long" because they are called by the Window Manager, which may be in a different bank than the program.

Upon return from `SetUpWindows`, `InitStuff` displays the mouse pointer with a `QuickDraw ShowCursor` call, then executes a `DrawWindows` subroutine to draw the content region of each window. To do this, `DrawWindows`

makes the window's GrafPort active by doing a SetPort, then calls the DrawWin subroutine to do the actual drawing.

Upon return from DrawWindows, InitStuff returns control to the Windows segment. Windows then calls PlayWithIt, the user's evnt loop. PlayWithIt is simply a loop that polls TaskMaster continually until the user presses Esc. I could have used GetNextEvent here, but why bother? TaskMaster monitors the mouse position and tells the program when the button has been pressed in a window control. With GetNextEvent, the program would have to do all that.

When the user finally presses Esc, PlayWithIt exits to the Windows segment. The only remaining job is to close the windows and shut down the tools.

CHAPTER 10

Menus

In the early days of personal computers, users could only interact with a program by pressing keys. The more polished programs presented a list, or *menu*, of the available choices; programs of the cruder variety forced the user to remember keyboard commands. All this changed when the mouse entered the scene — now the user could point at his or her menu choice and press the mouse button to select an item.

Because the Apple IIGS comes equipped with a mouse, your programs should take advantage of it whenever possible; that is, the program should display a *menu bar* at the top of the screen, and let the user select from menu items or “pull-down” submenus by pointing and pressing the mouse button. To use menus in your program, you need the services of the *Menu Manager*.

Menu Bars and Pull-Down Menus

A menu bar is a rectangular strip that displays a list of commands the user can activate by pressing the mouse button. Some of these commands may cause some predefined task to be performed. (For example, selecting “Cut” from the menu bar of a word-processing program generally removes a block of highlighted text from the screen.) More often, however, menu bar

commands are actually menu titles. Pressing the mouse button on a title causes a menu to drop down onto the screen. From now on, I'll refer to items on a menu bar as "titles," but be aware that they may represent menuless commands.

The System Menu Bar

Applications that use pull-down menus must present the titles of those menus in a *system menu bar* at the top of the screen. Menu titles will vary between programs, but the *Apple Human Interface Guidelines* has defined two of them, *File* and *Edit*, that commonly appear in application programs.

According to the *Guidelines*, selecting "File" should produce a menu that lets the user perform simple file-related operations. Typical commands in the File menu include New (open a new, untitled document), Save (store the document on disk), and Print. The File menu should also include the Quit command, which lets the user exit from the program.

Selecting the second standard menu title, "Edit," should produce a menu that provides commands for operating on objects. In a text-based program, for instance, the Edit menu may include Undo, Cut, Copy, and Paste commands.

The system menu bar may also include a special graphics object: the shape of an apple. (Perhaps I should say *the* apple, because this particular one is decked out in the logo colors of Apple Computer, Inc.) Selecting the apple calls up a menu of the available new desk accessories (NDAs). Usually, this menu also provides an *About* command that the user can select to learn more about using the program.

For example, in a program called MoneyMaker, the About command may read "About MoneyMaker . . ." Selecting it brings up an on-screen window with, perhaps, a copyright notice, the name and address of the program's publisher, instructions for using the program, and anything else the application wants to display. Of course, the window must also have an *OK* button that the user can press to make the window disappear.

You're probably wondering about this "button" business. After all, I haven't mentioned this feature before. There's a good reason for that: the About window is not an ordinary Window Manager window; it is a *dialog box* that has been produced by (you guessed it) the Dialog Manager.

Figure 10-1 shows a system menu bar equipped with the apple, File and Edit titles, and View and Special titles that produce menus pertinent to the menu bar's application program.

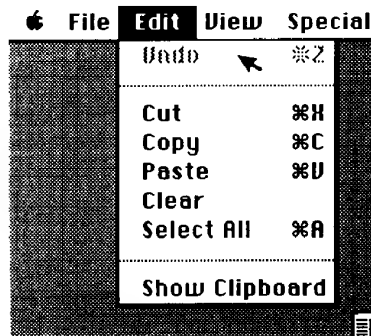


Figure 10-1

Pull-Down Menus

You have probably used pull-down menus already, and know how they work, so you may be tempted to skip this section. Don't! Although the way pull-down menus work is transparent to the user, and everything seems to take place automatically, the application program must do a lot of operations behind the scenes to make all this happen. For this reason, it's worthwhile to spend a little time investigating how pull-down menus operate. For the time being, I will describe their operation from a user's viewpoint. Later, I'll discuss them from a programmer's viewpoint.

When you point to a menu title and press the mouse button, the title text changes color and the title's menu drops down into a small window on the screen. Then, as you move the pointer down the menu (with the mouse button still down), each command is highlighted as the pointer reaches it; see Figure 10-2. Finally, when you reach the command you want, you release the mouse button. This makes the command name blink briefly and the menu disappear. When the program has finished executing the command, the menu title returns to its original color.

With all this happening, you may expect the programming for pull-down menus to be difficult. Relax. The Menu Manager (bless its heart) does much of the work, so the programming is rather easy (see later in this chapter).

Enabled and Disabled Menus

Sometimes you may want to prevent the user from selecting a certain menu or an item within it. For example, you may not want a game player to "Start a New Game" before he or she has told your program to "End This Game."

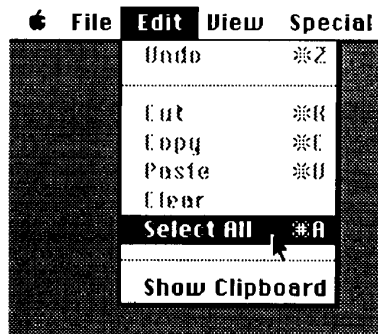


Figure 10-2

To prevent users from selecting an invalid menu title or menu item, you can disable it. When a menu title is disabled, it appears rather dim and fuzzy. The user can still pull its menu down, but every item will be dim and none of them will be selectable. Similarly, any disabled items in an enabled menu are also unselectable. Figure 10-3 shows an Edit menu with a disabled Undo command.

Menu Items

So far, I have referred to menu items as commands (words or phrases) the user can select by highlighting and releasing the mouse button — and indeed, that's what they usually are. However, you can also create menus that have a few variations, such as the one shown in Figure 10-4.

For example, you can divide groups of commands by function, by putting a dimmed line between them. A dividing line is itself a menu item, but since it is dimmed and disabled, the user can never select it. To him or her, it is simply a graphics object. The menu in Figure 10-4 uses dividing lines to separate the related commands Cut, Copy, and Paste from the commands Undo and Draw.

You can also precede a menu item with a *mark* (any character) to indicate some sort of status. For example, the menu in Figure 10-4 shows a crosshatch in front of the Draw command, to signify that the Draw feature is in effect. This program might also mark the Cut or Copy command when the user has cut or copied material but has not yet pasted it. The mark, then,

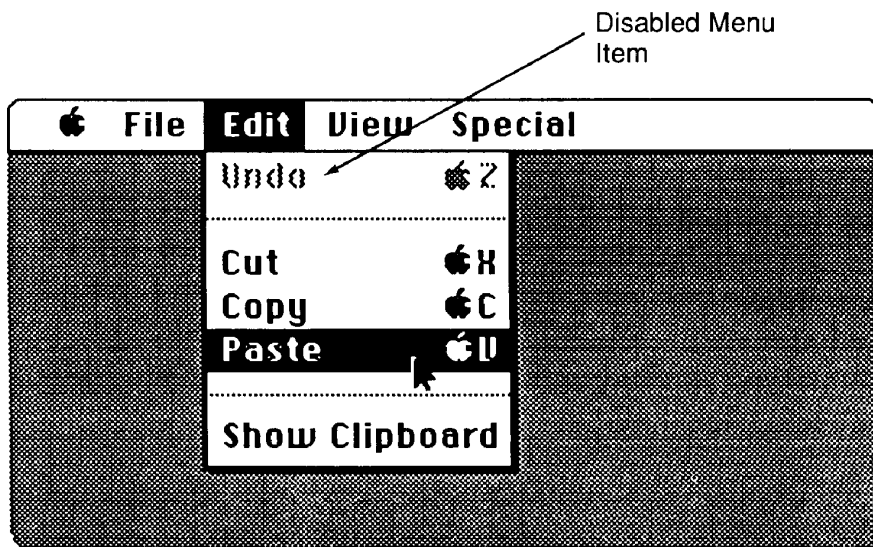


Figure 10-3

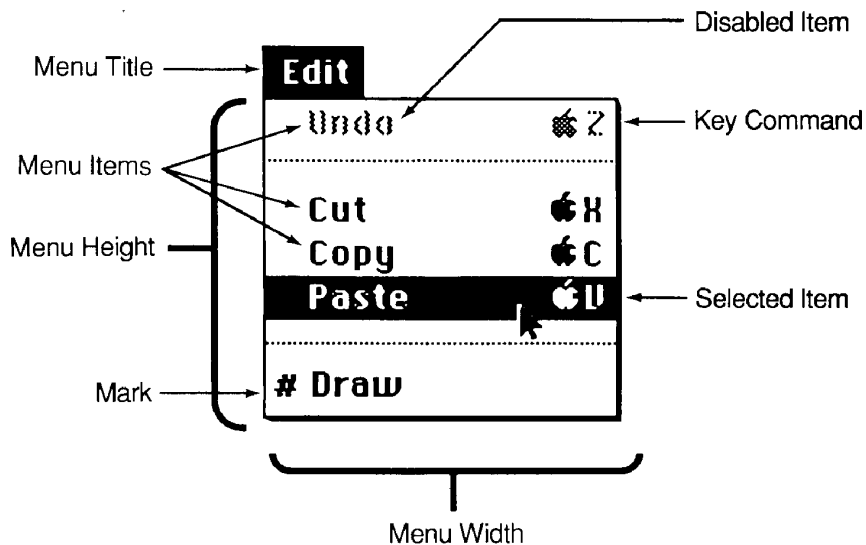


Figure 10-4

serves as a simple and effective way of telling the user about something without displaying a special message.

If there is a key equivalent for a menu item, your program can show it to the right of the command name. Generally, you should make the user hold down the OpenApple key when he or she presses the character key. This is indicated by showing an apple symbol ahead of the character. For example, the menu in the figure shows that the user can give an Undo, Cut, Copy, or Paste command by pressing OpenApple and Z, X, C, or V.

The letter keys for the Undo, Cut, Copy, and Paste commands weren't selected at random. They are the ones that Apple Computer recommends for use in *every* program that includes these operations. There are five more "standard" key combinations, for a total of nine. Here's the complete list:

Apple Menu

OpenApple-?	Help
-------------	------

File Menu

OpenApple-N	New
OpenApple-O	Open
OpenApple-S	Save
OpenApple-Q	Quit

Edit Menu

OpenApple-Z	Undo
OpenApple-X	Cut
OpenApple-C	Copy
OpenApple-V	Paste

Having a keyboard equivalent for Quit lets the user exit without selecting from a menu.

Creating Menu Bars and Menus

To equip your program with one or more menus, you first create the title bar, then insert the menu in it. To do this:

1. Initialize the Menu Manager with a MenuStartup call.

2. Define each menu and the items in it with a `NewMenu` call.
3. Add the menu to the system menu bar with an `InsertMenu` call.
4. Calculate the sizes of the menu bar and the menus with a `FixMenu-Bar` call.
5. Display the menu bar and the menu titles with a `DrawMenuBar` call.

At the end of this sequence, your menu bar is on the screen and the menus are in place (although they're invisible), ready to pull down and use. Table 10-1 summarizes these tool calls, in the order of their use.

Since I have just explained what these calls do, and the order in which they should appear in your programs, I won't bother describing them individually. However, I must explain the so-called menu/item line list that `NewMenu` requires.

The Menu/Item Line List

The `NewMenu` tool call sets up a menu title and the contents of items in the menu based on a list in your program. The length of this *menu/item line list* depends on the length of the menu.

Every menu/item line list is at least two lines long, where the first line defines the contents of the menu title (or command, in this case) and the second line contains an end-of-list, or *termination*, character. If the word or phrase in the menu bar is actually a menu title, the list must have one additional line for each item in the menu. Thus, the menu/item line list for a five-item menu must have seven lines: a title line, five item lines, and a termination line.

The title and item lines can contain up to five components, as follows:

1. A *start character*. You can use any characters you want, as long as the title and item characters are different.

I like to start title lines with a right angle bracket (>) and item lines with a space. The > symbol makes the list's title line easy to spot in a program listing (or search for using the editor), while the space character gives item lines a cleaner look than they would have with a "visible" character.

2. A one-character *place marker* in which `NewMenu` will store the length of the text string for the menu title or item name. I generally use L (for Length) for the marker, but any character will do. After all, the place marker doesn't *do* anything other than reserve space.

Table 10-1

<u>MenuStartup</u>		Start the Menu Manager
Call with:	PushWord <i>ProgramID</i> PushWord <i>DirectPageLoc</i> __MenuStartup	;Program ID ;Loc. of 1-page work area
Result:	None	
<u>NewMenu</u>		Create a new menu
Call with:	PushLong #0 PushLong <i>MenuStringPtr</i> __NewMenu	;Space for result ;Pointer to a menu/item line list
Result:	Handle of menu (long word); zero if an error occurred.	
Note:	See text for description of the menu/item line list.	
<u>InsertMenu</u>		Insert a menu in the current menu bar
Call with:	PushLong <i>AddMenu</i> PushWord #0 __InsertMenu	;Menu handle ;Insert at front of menu bar
Result:	None	
Note:	If you call InsertMenu immediately after NewMenu, the menu handle is already on the stack, and you can use the shorter form: PushWord #0 __InsertMenu	
<u>FixMenuBar</u>		Calculate sizes of menu bar and menus
Call with:	PushWord #0 __FixMenuBar	;Space for result
Result:	Height of menu bar (word).	
Note:	Normally you don't care about the height of the menu bar, and should follow FixMenuBar with a PLA instruction.	
<u>DrawMenuBar</u>		Display the completed menu bar
Call with:	__DrawMenuBar	
Result:	None	
<u>MenuShutDown</u>		Shut down the Menu Manager
Call with:	__MenuShutDown	
Result:	None	

3. The *text* for the menu title or item name, followed by a backslash (\) character.
4. The character N followed by the *identification number* of the menu or item. Working from left to right along the menu bar, menus are numbered from 1 to 255, while items in the program are numbered from 256 to 65534(!).
5. Optional *menu modifiers* (to be discussed shortly).

6. A *line termination character*; either a Return (decimal 13) or a Null (decimal 0).

The termination line at the end of the list contains only a termination character. This can be any character except the one you use to start item lines. A period (.) is a reasonable choice here.

For example, the following menu/item line list defines a menu that has only one item:

```
Menu1  dc c'>L File \N1',il'13' ;Title line
        dc c' LQuit\N256',il'13'   ;Item line
        dc c'.'
```

The N1 on the first line indicates that this menu title is to be inserted at the leftmost position on the spacebar.

Menu Modifiers

As just mentioned, the identification number on a title or item line may be followed by one or more menu modifiers. As Table 10-2 shows, there are modifiers that let you specify an item's text style (emboldened, underlined, or italicized), and show a marking symbol to the left of the name and a key command to the right of it. Note that you can use any of the modifiers to define an item, but only D and X to define a title.

If a menu item has an equivalent *key command*, you should remind the user of it by displaying the key or key combination to the right of the item name. To specify a key command for an item, include the * modifier in its

Table 10-2

Modifier	Action
*Kk	Display a key equivalent for the menu item, where <i>K</i> is the primary character and <i>k</i> is the secondary character.
B	Display the menu item text in bold type.
C <i>m</i>	Precede (mark) the menu item text with an <i>m</i> character.
D	Disable the menu item or title (i.e., make it dim).
I	Display the menu item text in italics.
U	Underscore the menu item text.
V	Display a dividing line beneath this item.
X	Use color replace highlighting

definition. This makes the Menu Manager display an apple symbol to the left of your primary character. The apple tells the user that to activate this command from the keyboard, he or she must press the OpenApple key as well as the character key.

Generally, the primary character is an uppercase letter, so the secondary character is its lowercase counterpart. For example, entering **Rr** lets the user press either OpenApple-Shift-R or OpenApple-R to access the menu item. (The terms *primary* and *secondary* are somewhat misleading here. While the primary character is the one that is displayed on the screen, all but rank novices will press OpenApple key, instead of OpenApple-Shift key. In this case, then, the so-called secondary character is actually the one that will be used most often.)

If the key command has no secondary character (e.g., it is a number), you must enter a space after the primary character. For instance, “*2 ” establishes OpenApple-2 as the only valid key command for this item.

I will discuss the *X* modifier shortly. *B*, *I*, and *U* are self-explanatory; they embolden, italicize, or underline an item. Underlining is useful for saving space in a long menu, because it doesn't take up a line of its own. The Menu Manager provides `GetItemStyle` and `SetItemStyle` tool calls that let you read and change these attributes from your program.

C lets you mark an item name with a character of your choice, where crosshatch (#) and asterisk (*) are the most likely candidates. The Menu Manager provides a `SetItemMark` call that lets you mark or unmark an item name from within a program. It also has a `CheckItem` call that stamps an item with a check mark, the symbol the Control Panel shows to indicate a default setting.

D makes an item unselectable and shows it dimmed. You can enable and undim an item from your program by calling the Menu Manager's `EnableItem` tool. Similarly, you can disable an item by calling `DisableItem`.

V inserts a dividing line between this item and the next one. Thus, *V* is similar to *U* (underline), except it gives the menu a more orderly appearance; *U* tends to jam items together vertically.

Figure 10-5 shows a menu that has the features I have described so far. Its menu/item line list would have the form:

```
Menu4 dc c' >LText\N4',il'13'           ;Title line
      dc c' LUndo\N264NV',il'13'         ;Dimmed,
                                           ; dividing line
      dc c' LLeft\N265C#*L1',il'13'      ;Marked with #,
                                           ; L keys
```

Text	
Undo	
#Left	⌘ L
Centered	⌘ C
Right	⌘ R
Bold	⌘ B
<i>Italic</i>	⌘ I

Figure 10-5

```

dc c' LCentered\N266*Cc',il'13' ;C key command
dc c' LRight\N267U*Rr',il'13'   ;Underlined,
                                ; R keys
dc c' LBold\N268B*Bb',il'13'    ;Bold, B keys
dc c' LItalics\N269I*Ii',il'13' ;Italics,
                                ; I keys
dc c' '.'                        ;Termination
                                ; line

```

This list assumes that the Text menu is the fourth menu in the program (it is numbered N4) and that the preceding menus had a total of eight items (numbered 256 through 263).

You can also display the multicolored Apple symbol as a menu title; simply enter @ for the title and put X after the menu number. By convention, if you use an Apple menu, make it the first one on the menu bar. The following directives define an Apple menu with one item, "About this program . . ."

```

Menu1 dc c'>L\N1X',il'13'      ;Apple menu title line
      dc c' LAbout this program...\N256',il'13'
                                      ;Item line
      dc c' '.'                  ;Termination line

```

You can also specify a disabled dividing line as an item, by entering a hyphen (-) for the title and a D after the item number, as in:


```
dc c' L-\N277D' ,il'13' ;Dividing line
```

Responding to Menu Events

It's possible to use menus with just the Event Manager and the Menu Manager in place, and to poll for mouse-up and mouse-down events with the `GetNextEvent` call. However, as an Event Manager tool, `GetNextEvent` only records the mouse's location when the event occurred; it neither knows nor cares whether that position is relevant. Thus, with `GetNextEvent`, your program must do all the work of relating a mouse event to the pointer location. It's much easier to activate the Window Manager — even if your application doesn't use windows — and let its TaskMaster keep track of the mouse.

`GetNextEvent` also falls short in processing key commands. No matter which key the user presses (and regardless of whether OpenApple is also pressed), `GetNextEvent` generates a key-down event and makes your program decipher the key code. By contrast, TaskMaster intercepts Open-Apple key commands automatically and, with help from the Menu Manager, executes the routines they represent. Your program never gets involved with carrying out key commands.

Mouse Events

When the user presses the mouse button in the system menu bar, TaskMaster executes the Menu Manager's *MenuSelect* routine. If the mouse pointer is not in a menu title, *MenuSelect* tells TaskMaster to ignore it and TaskMaster returns 0 (the null event) on the stack, as if the button had never been pressed.

However, if the mouse pointer *is* in a menu title when the user presses the button, *MenuSelect* highlights the title and pulls down its menu (if any). Then it puts the title's ID number in the high word of the task record's TaskData field and 0 in TaskData's low word. Of course, *MenuSelect* tells TaskMaster about the selection, but your program never hears about it. And why should it? After all, the user has simply moved the pointer to a selectable title; he or she has not yet *chosen* anything. To choose a menu item, you must *release* the mouse button while the item is selected (highlighted).

Hence, while the mouse button is being held down, TaskMaster calls *MenuSelect* repeatedly, but returns nulls to your program. The user's selections are a private matter between these two tools; the program needn't know about them.

If the user moves the pointer off the title and down to the first menu item, *MenuSelect* switches the highlighting from the title to the item and

stores the menu's ID in the high word (as before) and the item's ID in the low word. In fact, once the mouse button is down, MenuSelect doggedly monitors the pointer location. If the user moves the pointer to a new item, MenuSelect switches the highlighting and updates TaskData.

What happens when the user finally releases the button depends on where the mouse pointer is located? If the user has moved the pointer completely off the menu before he or she releases the button, MenuSelect says "Okay, false alarm," and takes the menu off the screen. It's then up to TaskMaster to inform your program where the button was released — in the system window, in the content region of an application window, or whatever. If the user releases the mouse button in a highlighted (selected) title or menu item, TaskMaster presents your program with task code 17, indicating "in system menu bar."

Responding to Mouse Events

Upon obtaining task code 17 from the stack, the program's event loop should use it as an index into a task table, just as the WINDOWS program in Chapter 9 does. Entry 17 in the table should point to a subroutine called, say, DoMenu that processes menu selections. DoMenu's job is to read what MenuSelect has stored in TaskData, act on it, then turn the menu's highlighting off and return to the event loop.

The content of DoMenu depends on how your program is using the menu bar. If any title is a stand-alone command (i.e., a title with no menu), DoMenu should read its ID number from the high word of TaskData and use that number to index into a table of command-processing subroutines. If your menu bar has four titles, DoMenu may look like Example 10-1.

On the other hand, if each of the words or phrases on the menu bar represents a menu title, your DoMenu subroutine can ignore the menu IDs in the high word of TaskData and work directly with the item IDs in TaskData's low word. Because item ID numbers start with 256, DoMenu should subtract 256 from the number (or simply clear bit 8) and use the result to index into your menu table — or item table, in this case. Example 10-2 shows the instructions you need here.

Key Events

Just as the TaskMaster works in conjunction with MenuSelect to keep track of mouse events for menus, it works with another Menu Manager routine, MenuKey, to check key-down events against key commands within the menus.

Example 10-1

```

DoMenu    START
          using GlobalData

          lda TaskData+2          ;Read the menu number
          asl a                  ; and double it
          tax

          jsr (MenuTable,x)       ;Go process the selection

          PushWord #0             ;Unhighlight the menu
          PushWord TaskData+2
          _HiliteMenu
          rts                    ; and return

MenuTable dc i'Ignore'           ;There is no Menu 0
          dc i'DoCommand1'
          dc i'DoCommand2'
          dc i'DoCommand3'
          dc i'DoCommand4'

          END

```

Example 10-2

```

DoMenu    START
          using GlobalData

          lda TaskData           ;Read the item number
          and #$00FF            ;Strip off the "256" bit
          asl a                  ; and double the result
          tax

          jsr (ItemTable,x)      ;Go process the selection

          PushWord #0            ;Unhighlight the menu
          PushWord TaskData+2
          _HiliteMenu
          rts                    ; and return

ItemTable dc i'DoItem1'
          dc i'DoItem2'
          dc i'DoItem3'
          dc i'DoItem4'
          ..
          ..

          END

```

When the user presses a key, TaskMaster does two things. First it stores the key's numeric code and the up/down state of the modifier keys (Control, Shift, Open-Apple) in the task record, then it sends this key information to

the MenuKey routine. MenuKey looks at this information to see whether the Open-Apple key was pressed. If not, it tells TaskMaster, “This has nothing to do with menu key commands,” and TaskMaster sends a key-down event to your program.

Otherwise, if the user pressed OpenApple as well as a character key, MenuKey checks whether that particular combination corresponds to a key command in any menu item. If MenuKey finds a matching item, it tells TaskMaster. TaskMaster, in turn, stores the menu’s ID number in TaskData and presents task code 17 (“in system menu bar”) to your program. The program doesn’t realize that a key command caused the event; it executes the item’s subroutine just as though the user had selected the item with the mouse.

This working partnership between TaskMaster and MenuKey lets your programs disregard key commands entirely. Unless you’re using keys for some other purpose, you can put an “ignore” in the task record’s key-down event entry.

Providing for New Desk Accessories

If your program has any menus at all, it should allow the user to call up any installed new desk accessories (NDAs). NDAs are kept in the DESK.ACCS subdirectory of the boot disk’s SYSTEM directory. The tool that provides for NDAs is one provided by the Desk Manager — FixAppleMenu. Details are:

```
_FixAppleMenu  Insert list of NDAs in the specified menu
                Call with:  PushWord MenuNumber      ;Menu number
                        _FixAppleMenu
                Result:  None
```

By convention, you should put the list of NDAs in the Apple menu. Since that is the first menu on the menu bar, your input to FixAppleMenu should be *PushWord #1*. The FixAppleMenu call belongs immediately before your FixMenuBar call.

An Example Program that Provides Menus

Example 10-3 lists a program called MENUS that’s an enhanced version of the WINDOWS program in Chapter 9. MENUS displays the same three

windows as before (although each now has a close box), but it also provides menus. The menu bar at the top of the screen has an Apple menu symbol and the titles File and Windows.

The Apple menu has only one item, *About this program*, but this item is disabled (dimmed). I will activate it in a later program.

The File menu also has only one item, Quit, which lets the user exit. Quit has the equivalent key command OpenApple-Q. The third menu, Windows, lets the user select a window and bring it to the front of the stack. This menu's main purpose is to let the user reinstate a window that he or she has closed previously.

By glancing through the listing, you will notice relatively few differences between WINDOWS and MENUS. The InitStuff segment now includes NewMenu and InsertMenu calls to build the menus, a Desk Manager FixAppleMenu call to install the list of NDAs in the Apple menu, a FixMenuBar call to size the menus, and a DrawMenuBar call to display the completed system menu bar. The task table in the EventLoop now has an "Ignore" for key-down events (KeyMenu will handle them automatically) and pointers to DoMenu and DoClose subroutines for system menu bar and go-away events (events 17 and 22, respectively).

When the user chooses a menu item, TaskMaster calls DoMenu, which reads the item's ID number from TaskData, strips off its "256" bit, and uses the result to call one of four subroutines, DoQuit, DoWin1, DoWin2, or DoWin3. DoQuit simply sets the QuitFlag and returns to EventLoop. The DoWin subroutines select a window (bring it to the front and make it active), then show it (make it visible if it is hidden).

The DoClose subroutine executes when the user has pressed the mouse button in the active window's close box. It simply calls HideWindow, to make the window disappear. (If it had called CloseWindow, the user wouldn't be able to redisplay the window, because it would no longer exist.)

There is also a new MenuData segment that holds the menu definitions. Note that the line for "About this program..." includes a D modifier to disable the item and that Quit's line includes a *Qq modifier, to display the OpenApple-Q key command. The rest of the program is the same as for WINDOWS, except that there's a minor change in WindowData. The window frame definition in each window parameter block now has a 1 in the second bit position, to install a close box.

Example 10-3

; MENUS displays three windows and lets the user shuffle, drag, or
 ; close them, or make them zoom or grow. It also provides three menus.
 ; The Apple menu has one item (About this program...), but it's disabled.
 ; The File menu has the Quit command and the Windows menu lets the
 ; user unhide a window and bring it to the front.

```

      absaddr on
      MCOPY Menus.macros

Menus      START
           using GlobalData

;-----
;
; Global equates used throughout the program.

ScreenMode    gequ $80           ;640 mode, no fill
MaxX          gequ 640          ;640 mode for Event Manager

           phk                  ;Set data bank to program
           plb                  ; bank to allow absolute addressing

           jsr InitStuff        ;Initialize everything
           bcs AllDone          ;Quit if initialization fails

           stz QuitFlag         ;Initialize the quit flag to 0
           jsr EventLoop        ;Let the user play

AllDone      anop               ;All is done, shut down
           PushLong Win1Ptr      ;Close the windows
           _CloseWindow
           PushLong Win2Ptr
           _CloseWindow
           PushLong Win3Ptr
           _CloseWindow

           _DeskShutDown         ;Desk Manager
           _MenuShutDown        ;Menu Manager
           _WindShutDown        ;Window Manager
           _CtlShutDown         ;Control Manager
           _EMShutDown          ;Event Manager
           _QDShutDown          ;QuickDraw II
           _MTShutDown          ;Miscellaneous Tools

           PushWord MyID         ;Discard the program's handle
           _DisposeAll

           PushWord MyID
           _MMShutDown          ;Memory Manager
           _TLShutDown          ;Tool Locator

           _Quit QuitParams      ;Do a ProDOS Quit call
           brk $F0              ;If it fails, break

      END

```

```

;*****
;
; Global Data
;
;*****

GlobalData    DATA

QuitParams    dc i4'0'                ;Return to caller
               dc i'$4000'            ;Make program restartable in memory

ToolTable     dc i'NumTools'          ;No. of tool sets in table
               dc i'4,$0100'          ;QuickDraw
               dc i'5,$0100'          ;Desk Manager
               dc i'6,$0100'          ;Event Manager
               dc i'14,$0100'         ;Window Manager
               dc i'15,$0100'         ;Menu Manager
               dc i'16,$0100'         ;Control Manager

TTEnd         anop

TableSize     equ TTEnd-ToolTable-2
NumTools      equ TableSize/4

TaskRecord    anop                    ;Buffer for task record
EventWhat     ds 2
EventMessage  ds 4
EventWhen     ds 4
EventWhere    ds 4
EventModifiers ds 2
TaskData      ds 4
TaskMask      dc i4'$01FFF'          ;Make TaskMaster do all it can

Win1Ptr       ds 4                    ;Window pointers
Win2Ptr       ds 4
Win3Ptr       ds 4

MyID          ds 2                    ;This will hold the program's i.d.

VolNotFound   equ $45                 ;ProDOS error

QuitFlag      ds 2                    ;Quit flag
               END

;*****
;
; InitStuff
;
; Initializes tool sets, gets space in bank 0 for use as direct
; page by tool sets that need it, and ensures that RAM-based
; tools are in memory.
;
;*****

InitStuff     START
               using GlobalData
               using MenuData

; Initialize the Tool Locator, Memory Manager, and Miscellaneous
; Tools.

```

344 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

```

    _TLStartup                ;Tool locator
    PushWord #0                ;Memory Manager
    _MMStartup

    pla                        ;Memory Manager returns program's ID
    sta MyID

    _MTStartup                ;Misc. Tools
    ldx #3
    jsr PrepareToDie

; Get some space for the direct page we need. QuickDraw needs
; three pages and each Manager except Window needs one page.

    PushLong #0                ;Space for handle
    PushLong #$600            ;Six pages
    PushWord MyID              ;Owner
    PushWord #$C005            ;Locked, fixed, fixed bank
    PushLong #0                ;Location
    _NewHandle
    ldx #$FF
    jsr PrepareToDie

    pla                        ;Read handle and store in dp
    sta 0
    pla
    sta 2

    lda [0]                    ;Get dp location from handle
    sta 4                      ; and store at loc 4 of dp

; Initialize QuickDraw

    pha                        ;Use dp obtained from handle
    PushWord #ScreenMode       ;Mode = 640
    PushWord #160               ;Max size of scan line (in bytes)
    PushWord MyID
    _QDStartup
    ldx #4
    jsr PrepareToDie

; Initialize Event Manager

    lda 4                      ;dp to use = QD dp +$300
    clc
    adc #$300
    sta 4
    pha
    PushWord #20                ;Queue size
    PushWord #0                 ;X clamp left
    PushWord #MaxX              ;X clamp right
    PushWord #0                 ;Y clamp top
    PushWord #200               ;Y clamp bottom
    PushWord MyID
    _EMStartup
    ldx #6
    jsr PrepareToDie

;-----
;
; Load the RAM-based tools I need.
```



```
LoadAgain      PushLong #ToolTable
                _LoadTools
                bcc ToolsLoaded
```

```
                cmp #VolNotFound
                beq DoMount
                sec
                ldx #$FE
                jsr PrepareToDie
```

```
DoMount        anop
                jsr MountBootDisk
                cmp #1
                beq LoadAgain

                sec
                rts
```

```
; The tools have been loaded. Initialize the Window Manager, Control
; Manager, Menu Manager, and Desk Manager.
```

```
ToolsLoaded     anop
                PushWord MyID          ;Window Manager
                _WindStartup
                ldx #14
                jsr PrepareToDie

                PushLong #0            ;Prepare screen for windows
                _RefreshDesktop

                PushWord MyID          ;Control Manager
                lda 4                  ;dp to use = EM dp + $100
                clc
                adc #$100
                sta 4
                pha
                _CtlStartup
                ldx #16
                jsr PrepareToDie

                PushWord MyID          ;Menu Manager
                lda 4
                clc
                adc #$100
                pha
                _MenuStartup
                ldx #15
                jsr PrepareToDie

                _DeskStartup           ;Desk Manager
```

```
;-----
;
; Build the menu bar by inserting the three menus (right to left order).
```

```
                PushLong #0            ;Windows menu
                PushLong #WindowsMenu
                _NewMenu
                PushWord #0
                _InsertMenu
```

```

        PushLong #0                ;File menu
        PushLong #FileMenu
        _NewMenu
        PushWord #0
        _InsertMenu

        PushLong #0                ;Apple menu
        PushLong #AppleMenu
        _NewMenu
        PushWord #0
        _InsertMenu

;-----
;
; Call the Desk Manager to install the list of NDAs in the system.

        PushWord #1                ;NDAs will go in Apple menu
        _FixAppleMenu

;-----
;
; Finish off getting the menu bar ready.

        PushWord #0
        _FixMenuBar
        pla                        ;Discard menu bar height

        _DrawMenuBar              ;Display the completed bar

        jsr SetUpWindows          ;Show the windows
        _ShowCursor              ; and the mouse pointer

        jsr DrawWindows           ;Draw the window contents

        clc                      ;Clear the carry flag
        rts                      ; and return

        END

;*****
;
; Set Up Windows
;
; Opens the three windows (but does not draw their contents).
;
;*****

SetUpWindows START
        using GlobalData
        using WindowData

        PushLong #0                ;Window 1
        PushLong #Win1ParamBlock
        _NewWindow

        pla
        sta Win1Ptr
        pla

        sta Win1Ptr+2

```

```

        PushLong #0                ;Window 2
        PushLong #Win2ParamBlock
        _NewWindow

        pla
        sta Win2Ptr
        pla
        sta Win2Ptr+2

        PushLong #0                ;Window 3
        PushLong #Win3ParamBlock
        _NewWindow

        pla
        sta Win3Ptr
        pla
        sta Win3Ptr+2

        rts
        END

;*****
;
; Event Loop
;
; Display windows until user chooses "Quit".
;
;*****

EventLoop      START
                using GlobalData

Again          lda QuitFlag
                bne NoMore

                PushWord #0          ;Space for result
                PushWord #$FFFF      ;Accept any event
                PushLong #TaskRecord ;Point to task record buffer
                _TaskMaster

                pla                  ;Is an event available?
                beq Again            ; No. Continue polling
                asl a                ; Yes. Double it
                tax                  ; and copy it into X.

                jsr (TaskTable,x)    ;Execute the event's routine,
                bra Again            ; then resume polling

NoMore         rts

TaskTable      anop                  ;Event Manager events
                dc i'Ignore'         ; 0 null
                dc i'Ignore'         ; 1 mouse down
                dc i'Ignore'         ; 2 mouse up
                dc i'Ignore'         ; 3 key down
                dc i'Ignore'         ; 4 undefined
                dc i'Ignore'         ; 5 auto-key
                dc i'Ignore'         ; 6 update
                dc i'Ignore'         ; 7 undefined
                dc i'Ignore'         ; 8 activate
                dc i'Ignore'         ; 9 switch

```

```

dc i'Ignore'           ; 10 desk acc
dc i'Ignore'           ; 11 device driver
dc i'Ignore'           ; 12 ap
dc i'Ignore'           ; 13 ap
dc i'Ignore'           ; 14 ap
dc i'Ignore'           ; 15 ap
dc i'Ignore'           ; 16 in desktop
dc i'DoMenu'           ; 17 in system menu bar
dc i'Ignore'           ; 18 in system window
dc i'Ignore'           ; 19 in window content region
dc i'Ignore'           ; 20 in drag region (title bar)
dc i'Ignore'           ; 21 in grow box
dc i'DoClose'          ; 22 in go-away region (close box)
dc i'Ignore'           ; 23 in zoom box
dc i'Ignore'           ; 24 in information bar
dc i'Ignore'           ; 25 in right scroll bar
dc i'Ignore'           ; 26 in bottom scroll bar
dc i'Ignore'           ; 27 in frame
dc i'Ignore'           ; 28 in drop region
END

```

```

;*****
;
; Ignore
;
; This do-nothing subroutine returns to the event loop for
; events we don't want.
;
;*****

```

```

Ignore    START
          rts
          END

```

```

;*****
;
; DoMenu
;
; Called when TaskMaster tells me that a menu item has been
; selected.
;
;*****

```

```

DoMenu    START
          using GlobalData

          lda TaskData           ;Get the item ID
          and #$00FF             ; and strip off the "256" bit
          asl a                   ;Double the result
          tax                     ; and copy it to X

          jsr (ItemTable,x)       ;Call the item's subroutine,

          PushWord #0             ; then unhighlight the menu
          PushWord TaskData+2
          _HiliteMenu

          rts

```

```

ItemTable      dc i'Ignore'           ;From Apple menu
               dc i'DoQuit'           ;From File menu
               dc i'DoWin1'           ;From Windows menu
               dc i'DoWin2'
               dc i'DoWin3'

               END

;*****
;
; Do Quit
;
; Sets the quit flag.
;
;*****
DoQuit          START
               using GlobalData

               lda #1
               sta QuitFlag
               rts

               END

;*****
;
; DoWin1
;
; Selects and shows window 1 in response to menu selection.
;
;*****
DoWin1          START
               using GlobalData

               PushLong Win1Ptr
               _SelectWindow

               PushLong Win1Ptr
               _ShowWindow

               rts

               END

;*****
;
; DoWin2
;
; Selects and shows window 2 in response to menu selection.
;
;*****
DoWin2          START
               using GlobalData

               PushLong Win2Ptr
               _SelectWindow

               PushLong Win2Ptr
               _ShowWindow

               rts

               END

```

350 APPLE II GS ASSEMBLY LANGUAGE PROGRAMMING

```
;*****  
;  
; DoWin3  
;  
; Selects and shows window 3 in repsonse to menu selection.  
;  
;*****  
  
DoWin3          START  
                using GlobalData  
  
                PushLong Win3Ptr  
                _SelectWindow  
  
                PushLong Win3Ptr  
                _ShowWindow  
  
                rts  
                END  
  
;*****  
;  
; DoClose  
;  
; Hides the active window when user presses button in its  
; close box.  
;  
;*****  
  
DoClose         START  
                using GlobalData  
  
                PushLong TaskData  
                _HideWindow  
  
                rts  
                END  
  
;*****  
;  
; Menu Data  
;  
;*****  
  
MenuData        DATA  
  
AppleMenu       dc c'>L\N1X',i1'13'  
                dc c' LAbout this program...\N256D',i1'13' ;Item disabled  
                dc c'.'  
  
FileMenu        dc c'>L File \N2',i1'13'  
                dc c' LQuit\N257*Qq',i1'13' ;Key command = Apple-Q  
                dc c'.'  
  
WindowsMenu     dc c'>L Windows \N3',i1'13'  
                dc c' LWindow 1\N258',i1'13'  
                dc c' LWindow 2\N259',i1'13'  
                dc c' LWindow 3\N260',i1'13'  
                dc c'.'  
                END
```

```

;*****
;
; Window Data
;
;*****

```

WindowData DATA

Win1ParamBlock anop

```

dc    i'Win1End-Win1ParamBlock'
dc    i2'%1101110110000000' Everything but info bar
dc    i4'Win1Title'            Pointer to window's title
dc    i4'0'                    Reserved (wRefCon)
dc    i2'26,0,190,620'        Zoomed content region
dc    i4'0'                    Default color table
dc    i2'0'                    Vertical origin
dc    i2'0'                    Horizontal origin
dc    i2'200'                  Data area height
dc    i2'640'                  Data area width
dc    i2'200'                  Max grow height
dc    i2'640'                  Max grow width
dc    i2'4'                    Number of pixels to scroll vertically.
dc    i2'16'                   Number of pixels to scroll

```

horizontally.

```

dc    i2'40'                   Number of pixels to page vertically.
dc    i2'160'                  Number of pixels to page horizontally.
dc    i4'0'                    Information bar text string.
dc    i2'0'                    Info bar height
dc    i4'0'                    Routine to draw shape (none, standard)
dc    i4'0'                    Routine to draw info. bar.
dc    i4'DrawWin1'             Routine to draw content.
dc    i'40,20,100,360'        Size/pos of content
dc    i4'-1'                   Window's order (-1 means topmost)
dc    i4'0'                    Window Manager allocates wind. record

```

Win1End anop

Win2ParamBlock anop

```

dc    i'Win2End-Win2ParamBlock'
dc    i2'%1101110110000000' Everything but info bar
dc    i4'Win2Title'            Pointer to window's title
dc    i4'0'                    Reserved (wRefCon)
dc    i2'26,0,190,620'        Zoomed content region
dc    i4'0'                    Default color table
dc    i2'0'                    Vertical origin
dc    i2'0'                    Horizontal origin
dc    i2'200'                  Data area height
dc    i2'640'                  Data area width
dc    i2'200'                  Max grow height
dc    i2'640'                  Max grow width
dc    i2'4'                    Number of pixels to scroll vertically.
dc    i2'16'                   Number of pixels to scroll

```

horizontally.

```

dc    i2'40'                   Number of pixels to page vertically.
dc    i2'160'                  Number of pixels to page horizontally.
dc    i4'0'                    Information bar text string.
dc    i2'0'                    Info bar height
dc    i4'0'                    Routine to draw shape (none, standard)
dc    i4'0'                    Routine to draw info. bar.
dc    i4'DrawWin2'             Routine to draw content.
dc    i'50,30,110,380'        Size/pos of content

```

```

        dc      i4'-1'                Window's order (-1 means topmost)
        dc      i4'0'                 Window Manager allocates wind. record
Win2End  anop

Win3ParamBlock anop
        dc      i'Win3End-Win3ParamBlock'
        dc      i2'%1101110110000000' Everything but info bar
        dc      i4'Win3Title'         Pointer to window's title
        dc      i4'0'                 Reserved (wRefCon)
        dc      i2'26,0,190,620'      Zoomed content region
        dc      i4'0'                 Default color table
        dc      i2'0'                 Vertical origin
        dc      i2'0'                 Horizontal origin
        dc      i2'200'                Data area height
        dc      i2'640'                Data area width
        dc      i2'200'                Max grow height
        dc      i2'640'                Max grow width
        dc      i2'4'                 Number of pixels to scroll vertically.
        dc      i2'16'                Number of pixels to scroll
horizontally.
        dc      i2'40'                 Number of pixels to page vertically.
        dc      i2'160'                Number of pixels to page horizontally.
        dc      i4'0'                 Information bar text string.
        dc      i2'0'                 Info bar height
        dc      i4'0'                 Routine to draw shape (none, standard)
        dc      i4'0'                 Routine to draw info. bar.
        dc      i4'DrawWin3'           Routine to draw content.
        dc      i'60,40,120,400'      Size/pos of content
        dc      i4'-1'                Window's order (-1 means topmost)
        dc      i4'0'                 Window Manager allocates wind. record
Win3End  anop

Win1Title      str 'Window 1'
Win2Title      str 'Window 2'
Win3Title      str 'Window 3'

PenPat1  dc h'11 11 11 11 11 11 11 11' ;Dithered blue pen pattern
        dc h'11 11 11 11 11 11 11 11'
        dc h'11 11 11 11 11 11 11 11'
        dc h'11 11 11 11 11 11 11 11'

Rect      dc i'20,20,40,40'           ;Rectangle painted in windows

END

;*****
;
; Draw Windows
;
; Fills in the content region of each window, working rearward.
;
;*****

DrawWindows  START
              using GlobalData

              PushLong Win3Ptr          ;Window 3
              _SetPort

              jsl DrawWin3

```



```

        PushLong Win3Ptr
        _ShowWindow

        PushLong Win2Ptr          ;Window 2
        _SetPort

        jsl DrawWin2

        PushLong Win2Ptr
        _ShowWindow

        PushLong Win1Ptr          ;Window 1
        _SetPort

        jsl DrawWin1

        PushLong Win1Ptr
        _ShowWindow

        rts
        END

;*****
;
; Draw Window 1
;
; Draws the contents of Window 1--a blue box on a white
; background. Called by DrawWindows initially and by the Window
; Manager when it updates the window.
;
;*****

DrawWin1 START
    using WindowData

        PushWord #3                ;Background is white
        _SetSolidBackPat

        PushLong #PenPat1          ;Set pen pattern to blue
        _SetPenPat

        PushLong #Rect             ; and paint the rectangle
        _PaintRect

        rtl                        ;RTL required for update events
        END

;*****
;
; Draw Window 2
;
; Draws the contents of Window 2--a white box on a green
; background. Called by DrawWindows initially and by the Window
; Manager when it updates the window.
;
;*****

DrawWin2 START
    using WindowData

```

354 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

```

        PushWord #2                ;Background is green
        _SetSolidBackPat

        PushWord #3                ;Set pen pattern to white
        _SetSolidPenPat

        PushLong #Rect             ; and paint the rectangle
        _PaintRect

        rtl                        ;RTL required for update events
        END

;*****
;
; Draw Window 3
;
; Draws the contents of Window 3--a green box on a red
; background. Called by DrawWindows initially and by the Window
; Manager when it updates the window.
;
;*****

DrawWin3 START
        using WindowData

        PushWord #1
        _SetSolidBackPat

        PushWord #2                ;Set pen pattern to green
        _SetSolidPenPat

        PushLong #Rect             ; and paint the rectangle
        _PaintRect

        rtl                        ;RTL required for update events
        END

*****
*
* Prepare To Die
*
* Dies if carry is set.
*
* Error number to display is in X register.
* Assumes that Miscellaneous Tools is active.
*
*****

PrepareToDie START
        bcs RealDeath             ;Carry = 1?
        rts                       ; No. Return to caller

RealDeath      phx                 ; Yes. Goodbye, program.
               PushLong #DeathMsg
               _SysFailMgr

DeathMsg       str 'Could not handle error '

               END

```

```

;*****
;
; Mount Boot Disk
;
; Tells the user to insert the disk that contains the RAM-based
; tools.
;
;*****

MountBootDisk  START

    _Set_Prefix SetPrefixParams
    _Get_Prefix GetPrefixParams

    PushWord #0                ;Space for result
    PushWord #195              ;Column position for dialog box
    PushWord #30               ;Row position for dialog box
    PushLong #PromptStr        ;Prompt at top of dialog box
    PushLong #VolStr           ;Volume name string
    PushLong #OKStr            ;String in Button 1
    PushLong #CancelStr        ;String in Button 2
    _TLMountVolume

    pla                        ;Obtain the button number
    rts                        ; and return to caller

PromptStr    str 'Please insert the disk.'
VolStr       ds 16
OKStr        str 'OK'
CancelStr    str 'Shutdown'

GetPrefixParams dc i'7'
               dc i4'VolStr'

SetPrefixParams dc i'7'
               dc i4'BootStr'

BootStr       str '*/'

END

```

CHAPTER 11

Controls

In Chapter 9 you learned about the zoom box, scroll box, close box, and other controls you can provide in the frame of a window. By using the resources of the *Control Manager*, you can also provide controls inside a window — that is, within its content region.

This chapter is a little different from preceding chapters. I will describe the Control Manager's controls (at least the predefined ones), but I won't say much about specific tool calls, nor will I provide a programming example that uses them. Indeed, this chapter is rather short — for a very good reason. I have kept the Control Manager details to a minimum because most people won't use it, at least not directly. Instead, programmers generally access the resources of the *Dialog Manager* to do control-related operations.

When controls are involved, the Dialog Manager calls on the Control Manager behind the scenes to do the necessary work. The Dialog Manager also does some of the tasks these controls require, such as scrolling when the mouse button is pressed in a scroll bar; tasks a Control Manager-based program would have to do manually. In short, using the Dialog Manager (instead of the Control Manager) for operations that involve controls saves you from a lot of unnecessary work. It's somewhat like the benefits you gain by using the Window Manager's TaskMaster instead of the Event Manager's GetNextEvent tool.

The Apple Human Interface Guidelines are very clear as to what certain controls should do, and I will abide with the Guidelines' recommendations throughout this chapter. (Indeed, I have attempted to do so throughout this entire book.) Therefore, when I say a certain control does something or other, you can mentally follow my statement with the phrase "in accordance with the Apple Human Interface Guidelines."

Predefined Controls

The Control Manager provides four types of predefined controls. Three of them — buttons, check boxes, and radio buttons — are on/off controls (see Figure 11-1). The fourth is the familiar scroll bar.

Buttons

A *button* makes something happen the instant you click or press it with the mouse. Buttons appear as round-cornered rectangles with a title inside.

The Apple Human Interface Guidelines suggest that if you want to include buttons, you provide at least two of them: an "OK" (or "Begin", or something similar) to proceed with the operation and a "Cancel" (or perhaps "Quit") to nullify it. Note that the OK button in Figure 11-1 has a thick outline, to indicate that the user may press the Return key, instead of the mouse button, to proceed with the operation.

While you may not realize it, you have already seen Control Manager buttons in action. The Program Launcher provides four of them — Disk, Open, Close, and Quit — when you boot the Apple IIGS System Disk. It gives the Open button a thick outline, which lets you press Open to run the program whose name is highlighted.

The controls described next, check boxes and radio buttons, are used to specify options for the action the OK button will perform.

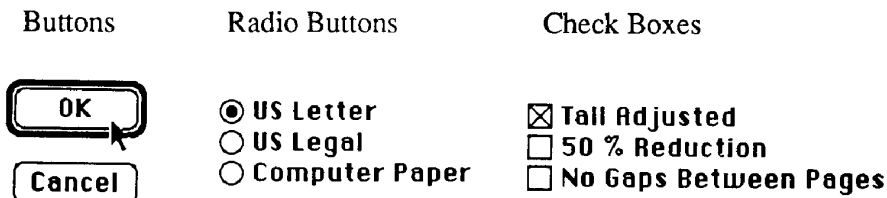


Figure 11-1

Check Boxes

A *check box* is a small square with a label to the right of it. The box contains an “X” when it has been selected (or checked); otherwise, it is blank (unchecked). Providing a group of check boxes allows the user to choose one or more options for some upcoming event.

Radio Buttons

Radio buttons are also on/off controls, but only one of them may be “on” (selected) at any given time — just like the buttons on a car radio. A radio button looks like a round check box. However, when selected, it contains a small black circle instead of an X.

Scroll Bars

The Control Manager’s scroll bars are the same as the Window Manager’s; they can have arrows, a scroll bar (or *thumb*), and gray “page” regions. However, while the scroll bars in the Window Manager’s so-called document windows only control what’s displayed in the content region, a Control Manager scroll bar can be used to control anything you want.

Scroll bars are particularly handy for letting users examine a list of data. Indeed, the System Disk’s Program Launcher provides one for just that purpose; to let you scroll through the list of available files and directories.

Scroll bars can also function as slide-type selectors, where the user can increase or decrease a program value by pressing the mouse. Slide selectors can be convenient in simulation games, for example, for changing the speed of a race car the user is “driving” or a jet plane or spaceship he or she is “flying.”

Finally, scroll bars can also serve as indicators, to show the user a particular program value in graphic form. They could, for example, be employed as the bars in a bar chart.

The Control Manager provides for a variety of controls under the broad category of *dials*. The scroll bar is the only predefined dial, but you can also (with a little more work) produce on-screen fuel gauges, thermometers, or just about any other kind of control or indicator.

Scroll Bar Components

When you add a scroll bar to a window’s frame using the Window Manager, it always comes decked out with arrows, a thumb, and (unless the content region shows the entire data area) “page” regions. However, because a

Control Manager scroll bar is a form of dial, you can equip it with as few or as many components as you want.

Figure 11-2 shows the components of scroll bars and their standard Control Manager terms. The nomenclature looks a little strange when applied to the horizontal scroll bar. While we naturally think in terms of left and right arrows, and page left and page right regions, the Control Manager thinks solely in terms of “up” and “down.” It only cares about a scroll bar’s parts, not whether the bar is vertical or horizontal.

Active and Inactive Controls

Like menu titles and items, controls may be active or inactive. An *active* control can be selected by pressing the mouse button inside it. Generally, this also highlights it (see Figure 11-3). For controls that have multiple parts, the highlighting only affects the part that the mouse pointer is in. For example, when the user presses the mouse button in an arrow of a scroll bar, only the arrow becomes highlighted.

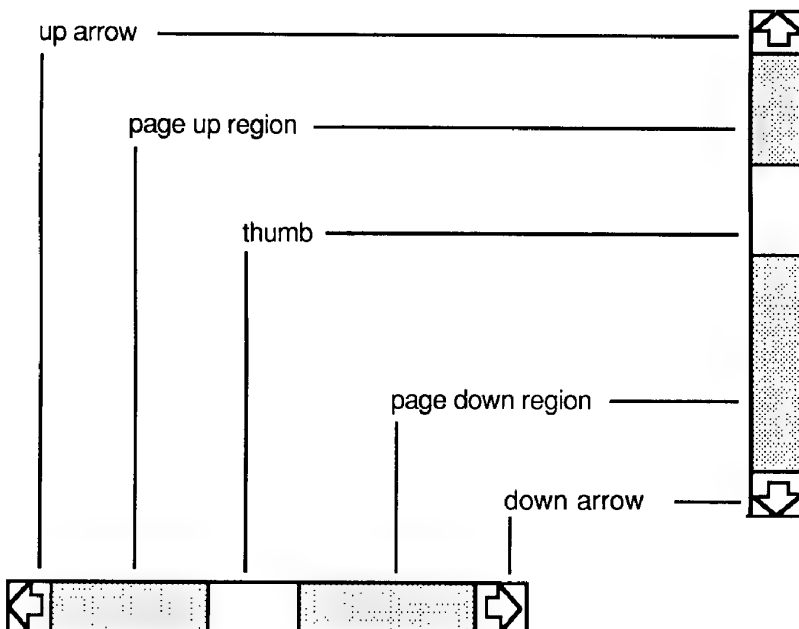


Figure 11-2

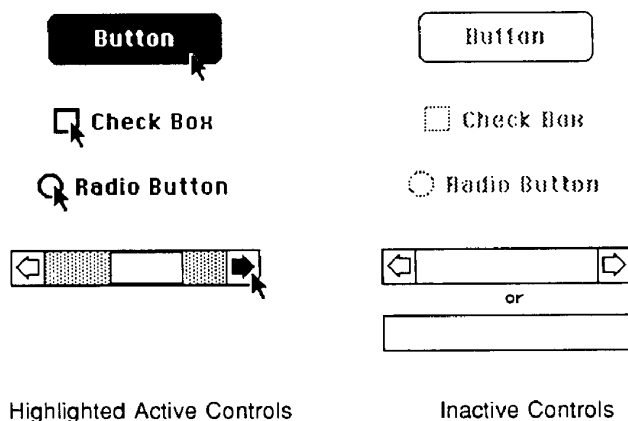


Figure 11-3

A control is made inactive when its operation is meaningless in the current context. For example, the IIGS Program Launcher starts with the Close button inactive, because in order to close a file, you must first open one. Inactive buttons, check boxes, and radio buttons appear dim on the screen. Inactive scroll bars look the same as active ones, but they're either lacking the thumb and paging regions or they're blank altogether (Figure 11-3 illustrates both states).

Control Manager Tool Calls

As I mentioned earlier, although the Control Manager is fully responsible for controls, you will probably find it easier to use its resources indirectly, through calls to the Dialog Manager. Thus, the only Control Manager calls that I must list are the ones that start it up and shut it down — `CtlStartup` and `CtlShutDown`; see Table 11-1. (I summarized these calls in Chapter 9, but they're worth reproducing here for completeness.)

Of course, the Control Manager also relies on other tool sets to help do its work. Before starting it, you must activate QuickDraw, the Event Manager, and the Window Manager. If your controls include text (e.g., buttons and check boxes), and you are *not* using the Dialog Manager to define them, you must also start the *Font Manager*. Refer to the Apple IIGS Toolbox Reference for details.

Table 11-1

__CtlStartUp		
Call with:	PushWord <i>ZeroPageLoc</i>	Start the Control Manager
	PushWord <i>ProgramID</i>	;Program ID
	PushWord <i>DirectPageLoc</i>	;Loc. of 1-page work area
	__CtlStartup	
Result:	None	
__CtlShutDown		
Call with:	__CtlShutDown	Shut down the Control Manager
Result:	None	

CHAPTER 12

Conducting Dialogs

It's possible to put controls and messages in the content region of a regular window, along with your picture or text, but this usually makes for a somewhat unprofessional looking program. Besides, if the window displays just a portion of an image, and lets the user scroll through the rest of it, your program would have to take care of scrolling the controls. That sounds like a lot of work — and it would be! It's much easier to create separate windows for interacting with the user and make them appear and disappear at appropriate times. The *Dialog Manager* can help you do this.

Through the services of the Dialog Manager, an application can communicate with users by displaying special-purpose windows that contain either a “dialog box” or an “alert box.” Both kinds of boxes usually contain OK and Cancel buttons, but they are very different otherwise.

A *dialog box* simply requests information from the user. For example, in a print operation, it may ask for the paper size (regular or legal) and type (single-sheet or continuous), and the name of the document to print. A dialog box is, then, a program's way of conducting a normal conversation, or dialog, with the user.

An *alert box* reports errors or issues warnings. While a dialog box generally appears as the result of something the user has done (e.g., choosing

“Print” in a menu), an alert box appears only when something has gone wrong or some unusual situation must be brought to the user’s attention.

Dialog Boxes

Figure 12-1 shows a typical dialog box, one in which the user presses buttons, checks boxes, and fills in blanks. Besides these items, a dialog box may contain controls (such as a close box, zoom box, or scroll bars), graphics images (icons or QuickDraw pictures), or anything else the application wants to put in it. The Dialog Manager provides for two kinds of dialog boxes, “modal” and “modeless.”

Modal Dialog Boxes

A modal dialog box is one that requires the user to respond before doing anything else. Once it appears, only pressing its OK or Cancel button makes it disappear; clicking the mouse button anywhere else just makes the speaker beep. In other words, the user is locked into the state or “mode” of having to respond to whatever the dialog box wants. In this respect, a modal dialog box is similar to an alert box. Note that the dialog box shown in Figure 12-1 is of the modal variety.

You can also set up modal dialog boxes that have no buttons at all. This is handy for displaying a “Please wait” type message while the application is performing a lengthy operation, such as sorting data, printing a document, or loading a file from disk.

Modeless Dialog Boxes

Unlike a modal dialog box, a modeless dialog box requires no attention whatsoever from the user. He or she can work in a document (i.e., Window Manager) window, select from menus, or do anything else the application allows, as if the dialog box isn’t even there! A modeless dialog box is, then, something the application provides strictly for the user’s convenience; he or she can use it or not.

A modeless title box can also take on the attributes of a regular document window. That is, an application can (and, in most cases, should) set up one that can be moved, made inactive or active, or closed like a document window. Figure 12-2 shows a modeless dialog box that a word-processing program might produce. Here, you could make the box disappear by clicking in its close box or, if it’s the active window, choosing Close from the file menu.

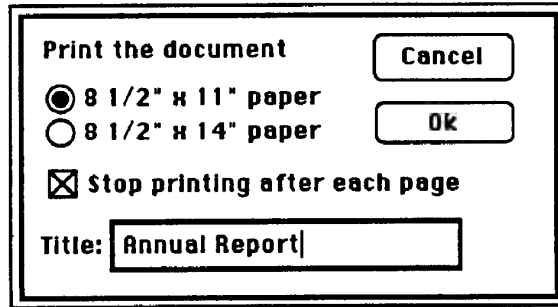


Figure 12-1

A modeless dialog box doesn't necessarily disappear when the user presses one of its buttons, however, as a modal box does. The application often keeps it around for future use. The Change box in Figure 12-2 is one that should be retained after the user presses a button; he or she may want to make more changes later, or maybe even right away.

Alert Boxes

As I mentioned at the beginning of this chapter, an alert box reports errors or issues warnings to the user when something significant has happened (or is about to happen) or something has gone wrong. Figure 12-3 shows an

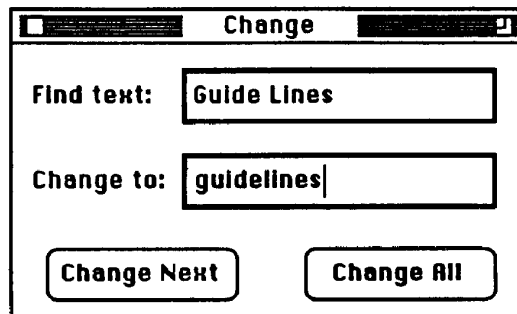


Figure 12-2

alert box that might appear when a user has selected a Quit command without first saving the active document.

Note that the *Don't Erase* button is the default here (it has a thick outline) — as it indeed should be! The Quit operation will be canceled if the user presses the Return key. Because alert boxes generally signal some drastic action, the button that lets the user back out of the operation should always be the default. It should choose “Don’t” rather than “Do”.

Types of Alert Boxes

There are three types of predefined alert boxes — Stop, Note, and Caution — each indicated by a unique icon at the top left-hand corner. Figure 12-3 shows the Caution icon. The icons for Stop and Note alerts are similar, but instead of an exclamation point, they show a hand and a “talking” face, respectively.

According to the *Apple Human Interface Guidelines*:

- A Stop alert box should signal a serious problem that requires remedial action by the user. One might appear when a disk is full or the user has removed a disk from a drive.
- A Note alert box should signal a minor mistake that wouldn’t produce disastrous consequences if left as is.
- A Caution alert box should indicate an operation that may or may not have undesirable results if it’s allowed to continue.

The Dialog Manager provides the tool calls `StopAlert`, `NoteAlert`, and `CautionAlert` to create these alert box types. Except for the fact that each call displays a different icon at the top left corner, they are identical. There is also a more general-purpose call, `Alert`, that doesn’t put any icon in the box; you can specify one of your own or omit it entirely.

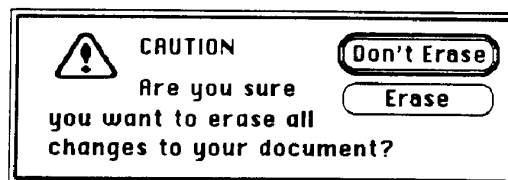


Figure 12-3

Stages of an Alert

If the user makes some minor mistake, such as trying to print a page whose number doesn't exist, you may want to start an alert sequence without displaying an alert box.

The Dialog Manager keeps track of four successive *stages* of an alert, and lets your program define a response for each one. That way, if a user persists in making the same mistake, the program can issue increasingly helpful (or, like a bill collector, increasingly sterner) messages. For example, you could make the first two occurrences of a mistake produce a beep and the next two bring up an alert box. This would be reasonable for a mistake that was probably accidental, such as choosing Cut when no block of text has been selected.

Programming Dialogs and Alerts

To include dialog or alert boxes in your program, begin by starting the Memory Manager, Desk Manager, QuickDraw, the Event Manager, Window Manager, Control Manager, and the Line Editor, in that order. The Line Editor, or LineEdit for short, requires a page of working space in bank 0. Allocate that space in your NewHandle call and start LineEdit with a sequence of the form:

```

PushWord MyID      ; Program ID from Memory Manager
lda 4              ; ZP to use = QD ZP + $500
clc
adc #$100
sta 4
pha
LEStartup
ldx #20
jsr PrepareToDie

```

That done, start the Dialog Manager with a DialogStartup call (see Table 12-1). Then, if your program uses menus, start the Menu Manager.

What the rest of the program contains depends on your application. If it includes dialog or alert boxes, you will have to create them and specify which items (buttons, scroll bars, text, etc.) belong in them.

Unlike windows and menus, however, dialog and alert boxes — or

Table 12-1

__DialogStartup		Start the Dialog Manager
Call with:	PushWord <i>ProgramID</i> ;Program ID	
	__DialogStartup	
Result:	None	
Note:	The Dialog Manager uses the Control Manager's direct page for working space.	
__DialogShutDown		Shut down the Dialog Manager
Call with:	__DialogShutDown	
Result:	None	
__LEStartup		Start the Line Editor
Call with:	PushWord <i>ProgramID</i> ;Program ID	
	PushWord <i>DirectPageLoc</i> ;Loc. of 1-page work area	
	__LEStartup	
Result:	None	
__LEShutDown		Shut down the Line Editor
Call with:	__LEShutDown	
Result:	None	

rather, the *windows* that hold them — are not necessarily created when the program starts. (You certainly shouldn't begin by confronting the user with an alert, for instance.) Instead, you can create these windows when they're needed. You may, for example, create a dialog window when the user chooses a menu item.

Once a dialog or alert window is active, your program must monitor the user's activities, to see what he or she has selected. Finally, you must remove a window from the screen when it's no longer needed.

The Dialog Manager provides tool calls for all these activities, and I will describe them shortly. But first I must discuss *item lists*, which are required to create both dialogs and alerts.

Item Lists

An item list is a summary of the specifications for a particular item in a dialog or alert box. It tells the Dialog Manager everything necessary to make that item an integral part of the box. An item list contains the following parameters:

- An *ID number* that identifies the item in the dialog. All subsequent tool calls that involve the item will refer to it by this number.
- A *display rectangle* that specifies the location of the item, in the local coordinates of the box.
- The *item type*; that is, button, check box, radio button, text, user-defined control, or whatever.
- An *item descriptor*. This is generally a pointer to a text string, such as the label in a button, the title beside a check box or radio button, or a message to the user.
- An *item value*; an initial value for the item.
- A *flag* that tells whether the item should be visible or invisible and, for some items, gives specific information (the group or “family” number of a radio button, whether a scroll bar is horizontal or vertical).
- The *color table* QuickDraw should use to draw the item.

Let’s take the parameters one at a time.

ID Number

The ID number parameter can range from 1 to 255, which means that you can put up to 255 items in a single dialog or alert box. The item numbered 1 becomes the *default*; the control that’s activated if the user presses Return. The Dialog Manager assumes item 1 is a button and gives it a bold outline. (After all, assigning 1 to anything except a button is meaningless.)

In general, item 1 should be the OK button and item 2 the Cancel button — or vice versa, in an alert box. Of course, if you don’t want a default button, don’t number any item 1.

Display Rectangle

The display rectangle, `ItemRect`, is an imaginary box in which the item is to be drawn. It specifies the location of the item, in the local coordinates of the box.

Item Type

The Dialog Manager provides for 12 different types of items, as summarized in Table 12-2.

Table 12-2

Type	Value	Description
ButtonItem	10	Simple button.
CheckItem	11	Check box.
RadioItem	12	Radio button.
ScrollBarItem	13	Special scroll bar for dialogs.
UserCtlItem	14	User-defined control.
StatText	15	Static text (up to 255 characters); text that cannot be edited.
LongStatText	16	Long static text (up to 32,767 characters).
EditLine	17	<i>Dialogs only.</i> A line of text that can be edited.
IconItem	18	<i>Dialogs only.</i> Icon.
PicItem	19	QuickDraw picture.
UserItem	20	<i>Dialogs only.</i> A user-defined item, such as a string that changes each time the box appears.
UserCtlItem2	21	Another user-defined control.

The Dialog Manager recognizes these types by numbers between 10 and 21, so you normally set up a list of the types your program uses with a series of Equates (equ's). A program that uses all 12 types would contain:

```

; Item types for Dialog Manager.

ButtonItem      equ 10    ;Button
CheckItem       equ 11    ;Check box
RadioItem       equ 12    ;Radio button
ScrollBarItem   equ 13    ;Scroll bar
UserCtlItem     equ 14    ;User-defined control
StatText        equ 15    ;Static text
LongStatText    equ 16    ;Long static text
EditLine        equ 17    ;Editable text
IconItem        equ 18    ;Icon
PicItem         equ 19    ;QuickDraw picture
UserItem        equ 20    ;User-defined item
UserCtlItem2    equ 21    ;Another user-defined control

```

In addition, if you add \$8000 to the value of an item type, the Dialog Manager will *disable* it; that is, make it unselectable. Text should be disabled (of course), but you may also want to disable a user-defined item, or an icon or QuickDraw picture. You may even want to disable a control, so the user can't do anything with it!

Unlike the Control Manager, the Dialog Manager does not provide a way to make a control inactive (i.e., show it dimmed). You can only disable it, and a disabled control looks just like an enabled one.

By convention, you would set up the \$8000 value as, perhaps,

```
ItemDisable equ $8000
```

and disable an item by specifying its type with the form:

```
EditLine+ItemDisable
```

Figure 12-4 shows an example of the various item types, with some of them disabled.

Having mentioned the EditLine item, I should mention the commands you can use to edit one. They are:

- Clicking in an item displays a blinking vertical line, to show the place where text may be inserted.

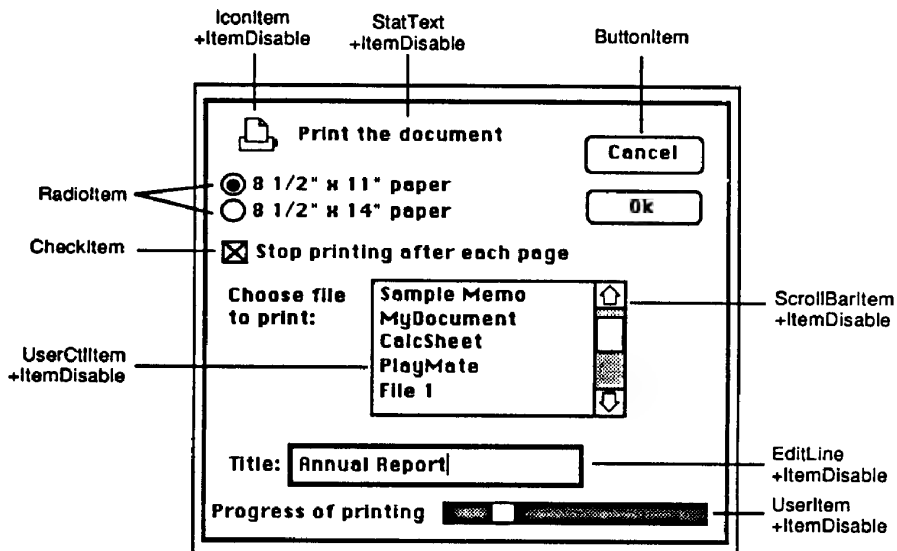


Figure 12-4

- Pressing the left or right arrow key moves the insertion point by 1 character. Pressing an arrow key with OpenApple down moves the insertion point to the beginning or end of the line. Pressing one with Option down moves the insertion point to the preceding or following word.
- Dragging over text selects it for replacement by whatever the user types. Double-clicking selects a word, while triple-clicking selects the entire line. Double-clicking followed by dragging extends or shortens the selection a word at a time.
- Pressing Delete deletes the current selection or the character preceding the insertion point.
- Control-F deletes the character that follows the insertion point, or the current selection.
- Control-Y deletes from the insertion point to the end of the line, or the current selection.
- Control-X deletes the entire line, or the current selection.
- OpenApple-X, -C, and -V do the same Cut, Copy, and Paste functions as they do in the regular editor.

Item Descriptor and Value

For a Button, Check, Radio, or StatText item, the item descriptor is a pointer to a text string. This string must be in standard ProDOS format, so you normally use the *str* macro to define it. For example,

```
OKButton str 'OK'
```

The item value for a Button or StatText item is zero, but for a Check or Radio item, make it either 0 (off) or 1 (on).

For a LongStatText (long static text) item, the item descriptor must point to the beginning of the string and the item value must contain the string's length in bytes. The following string uses ASCII carriage return (\$0D) characters to break the the text between lines:

```
VeryLongText  dc c'this text is really very
                long. ',h'0D'
                dc c'It just goes on . . . ',h'0D'
                dc c'and on . . . ',h'0D'
```

```

        dc  c'and on . . . ',h'OD'
        dc  c'and on . . . ',h'OD'
        . .
        . .
        dc  c'That's the beauty of using
            LongStatText:',h'OD'
        dc  c'You can produce many more',h'OD'
        dc  c'characters than with regular
            StatText.'
EndLongText    anop

```

The *anop* at the end lets you define the item value as

```
EndLongText-VeryLongText
```

For an *EditLine* item, the item descriptor points to a string that is to appear when the dialog window comes up on the screen and the item value is the maximum allowed length of the string. For example, if the user is to enter a ProDOS filename, you could limit the string to 15 characters and define the default text with, perhaps,

```
EditLString    str    'Untitled'
```

To omit the default text, set the item descriptor to 0.

Item Flag

The item flag is a word-size value that is only meaningful for buttons (both simple and radio) and scroll bars. Set it to 0 for any other item type.

For a simple button, the flag can only have two values: 0 to draw a round-cornered button or 2 to draw one with square corners. For a radio button, the flag tells whether the button is visible and assigns it to a “family” (see Figure 12-5). Since turning on one radio button turns off all others in a family, the flag lets the Dialog Manager know which buttons are related. A scroll bar’s item flag (also shown in Figure 12-5) assigns its controls and specifies whether the bar is horizontal or vertical.

Tool Calls for Dialog Boxes

Table 12-3 shows the most useful tool calls for dialog boxes. In general, you would call them in the order they’re listed; that is, you would call

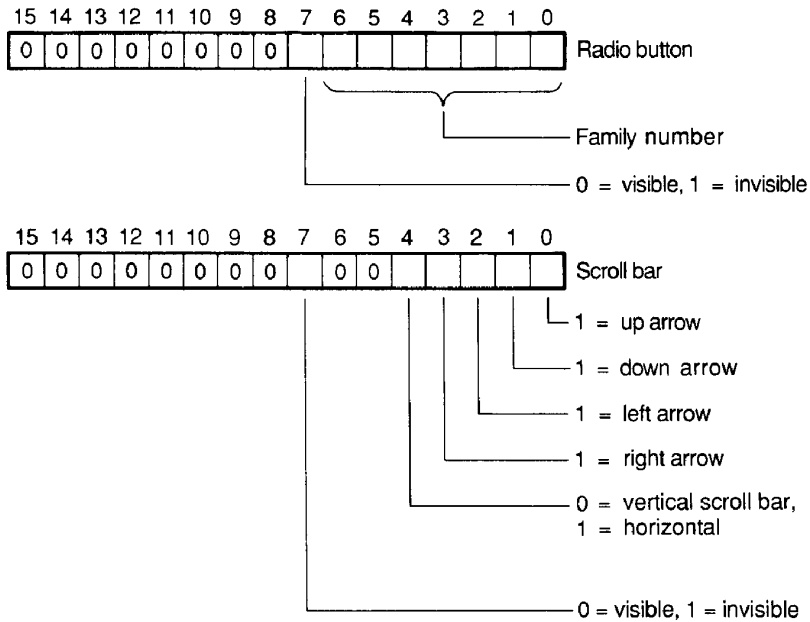


Figure 12-5

NewModalDialog or NewModeLessDialog to create any dialogs you need, then call NewItem for each item you want to add to the dialog. Finally, when you no longer need a dialog, you would call CloseDialog to get rid of it.

Table 12-3

Creating and Disposing of Dialogs		
<u>NewModalDialog</u>		Create a new modal dialog box
Call with:	PushLong #0	;Space for result
	PushLong <i>dBoundsRect</i>	;Pointer to window rectangle
	PushLong <i>dVisible</i>	;1 if visible, 0 if not
	PushLong <i>dRefCon</i>	;Any value you'd like to associate with this window
<u>NewModalDialog</u>		
Result:	Pointer to dialog's port (long word)	
Note:	The top coordinate should be at least 25 points below the top of the screen, to allow for the menu bar.	

Table 12-3 (cont.)

Creating and Disposing of Dialogs (cont.)		
<hr/>		
__NewModelessDialog	Create a new modeless dialog box	
Call with:	PushLong #0	;Space for result
	PushLong <i>dBoundsRect</i>	;Pointer to window rectangle
	PushLong <i>dTitle</i>	;Pointer to title string (0 for no title)
	PushLong <i>dBehind</i>	;Pointer to window the dialog should be behind;
		; - 1 puts it in front
	PushWord <i>dFlag</i>	;Flags describing the window's frame (see
		; Figure 9-5)
	PushLong <i>dRefCon</i>	;Any value you'd like to associate with this
		; window
	PushLong <i>FullSize</i>	;Pointer to Rect defining window's zoomed size
	__NewModelessDialog	
Result:	Pointer to dialog's port (long word)	
__CloseDialog	Close a dialog window	
Call with:	PushLong <i>theDialog</i>	;Pointer to dialog's port
	__CloseDialog	
Result:	None	
Note:	This is simply the Dialog Manager's equivalent of the Window Manager's CloseWindow call.	

Creating and Removing Items		
<hr/>		
__NewItem	Add a new item to the dialog's item list	
Call with:	PushLong <i>theDialog</i>	;Pointer to dialog's port
	PushWord <i>ItemID</i>	;ID number (1 to 255)
	PushLong <i>ItemRect</i>	;Pointer to display rectangle
	PushWord <i>ItemType</i>	;See table 12-2
	PushLong <i>ItemDescr</i>	;Item descriptor (e.g., string pointer)
	PushWord <i>ItemValue</i>	;Item value
	PushWord <i>ItemFlag</i>	;See figure 12-5 and related text
	PushLong <i>ItemColor</i>	;Pointer to item's color table (0 fo default)
	__NewItem	
Result:	None	
Note:	See "Item Lists" for a description of these inputs.	
__RemovedItem	Remove an item from the dialog	
Call with:	PushLong <i>theDialog</i>	;Pointer to dialog's port
	PushWord <i>ItemID</i>	;ID number of item
	__RemovedItem	
Result:	None	

Handling Dialog Events		
<hr/>		
__Modal Dialog	Handle modal events	
Call with:	PushWord #0	;Space for result
	PushLong <i>filterProc</i>	;Pointer to a filter procedure
	__ModalDialog	
Result:	ID number of item hit (word).	

Table 12-3 (cont.)

Handling Dialog Events (cont.)		
<hr/>		
__IsDialogEvent		Check for modeless events
Call with:	PushWord #0	;Space for result
	PushLong <i>theEvent</i>	;Pointer to the event record
	__IsDialogEvent	
Result:	A word. Nonzero if <i>theEvent</i> is a dialog event	
<hr/>		
__DialogSelect		Handle modeless event
Call with:	PushWord #0	;Space for result
	PushLong <i>theEvent</i>	;Pointer to the event record
	PushLong <i>theDialog</i>	;Pointer to variable at which to store the dialog
		; pointer
	PushLong <i>itemHit</i>	;Pointer to an integer at which to store the
		; item number
	__DialogSelect	
Result:	A word. Nonzero if the event involves an enabled dialog item.	
<hr/>		
Get Text		
__GetText		Get text from dialog box
Call with:	PushLong <i>the Dialog</i>	;Pointer to the dialog
	PushLong <i>itemID</i>	;ID number of item
	PushLong <i>theString</i>	;Pointer to a buffer to put the text in
	__GetText	
Result:	None	
Note:	Space for the string must be allocated before calling GetText.	

Handling Modal Events

To handle events in a modal dialog, simply call `ModalDialog` immediately after displaying the dialog box. If the front window is a modal dialog, `ModalDialog` monitors it and handles any events that occur. If an event is an enabled dialog item, it returns with the item's ID on the stack. If you dialog has only one enabled item (say, only an OK button), you can discard the ID and close the window with `CloseDialog`. Otherwise, you must use the ID to determine which item caused the "hit."

Note that `ModalDialog` takes a single input: a pointer to a "filter" procedure. If you push a 0 onto the stack, `ModalDialog` uses a default filter procedure in which it returns the ID of the default button (1) if the user presses Return. The *Apple IIGS Toolbox Reference* has full details on designing your own filter procedures, in case you're interested.

Handling Modeless Events

Handling events in a modeless dialog is a little trickier, in that it involves two calls: `IsDialogEvent` to check for modeless events and `DialogSelect` to actually handle the event. Both calls belong in your event loop, immediately following the `TaskMaster` calls.

`IsDialogEvent` takes one input: a pointer to the same event record the `TaskMaster` uses. The word it returns on the stack has a nonzero value (for TRUE) if the event has occurred in a dialog window; otherwise, the word contains zero (for FALSE). A nonzero result is your signal to pass the event to `DialogSelect`.

While `IsDialogEvent` simply indicates whether a dialog event has occurred, `DialogSelect` tells your program to the event record (same as for `IsDialogEvent`), a pointer to a 2-word memory location that will hold the dialog pointer, and a pointer to a 1-word location that will hold the item number. `DialogSelect` returns the same kind of TRUE/FALSE indicator as `IsDialogEvent`. Here, TRUE indicates that the event involves an enabled item.

Like the Window Manager's `TaskMaster`, `DialogSelect` can take care of some events itself. Specifically:

- If the event is an activate or update for a dialog window, `DialogSelect` activates or updates the window and returns FALSE.
- If the event is a key-down or auto-key event and an `EditLine` item is enabled, `DialogSelect` returns TRUE. In the absence of an enabled `EditLine`, key-down and auto-key events generate a FALSE result.
- If the mouse button is released inside an enabled control, `DialogSelect` returns TRUE; otherwise, it returns FALSE.
- If the mouse button is pressed in any other enabled item (e.g., an icon), `DialogSelect` returns TRUE. For mouse-down events in any disabled item or in no item, `DialogSelect` returns FALSE.

In short, then, your program must respond to TRUE results and ignore FALSE ones. Example 12-1 shows the kind of code needed to deal with modeless dialog events. To keep things simple, I'm assuming the application has only one modeless dialog window.

Get Text

The final tool call, `GetIText`, is a handy one. It lets you read the text of an `EditLine` in a dialog box. After all, if you're allowing the user to enter

something, your program must have a way of *reading* what he or she has typed.

Example 12-1

```

Again      anop

           lda QuitFlag      ;Quit flag still zero?
           bne AllDone       ; No. Exit

           PushWord #0
           PushWord #-1
           PushLong #EventRecord
           _TaskMaster

           pla                ;Has an event occurred?
           beq CheckDM        ; Maybe so. Check with the Dialog Mgr.

           asl a              ; Yes. Handle it
           tax
           jsr (TaskTable,x)
           bra Again

CheckDM     PushWord #0
           PushLong #EventRecord
           _IsDialogEvent

           pla                ;Dialog event?
           beq Again          ; No. Continue polling

           PushWord #0        ; Yes. Read it
           PushLong #EventRecord
           PushLong #theDialog
           PushLong #ItemHit
           _DialogSelect

           pla                ;Did DialogSelect deal with it?
           beq Again          ; Yes. Continue polling

           lda ItemHit        ; No. Application must handle it
           asl a
           tax

           jsr (DEventTable,x)

           bra Again
           rts

theDialog  ds 4              ;Space for dialog pointer
ItemHit    ds 2              ;Space for item number

; Dialog event table

DEventTable dc i'Ignore'    ;There's no item 0
            dc i'DoItem1'   ;Item 1
            dc i'DoItem2'   ;Item 2
            (etc.)

```

Example Dialog Box Program

Example 12-2 shows a program called MODALD that displays a modal dialog box when the user selects "About this program . . ." from an Apple menu on the menu bar. Note that MODALD is simply an enhanced version of MENUS with code added for the menu and dialog box.

Example 12-2

```
; MODALD is an enhanced version of MENUS in which choosing "About this
; program..." from the Apple menu brings on a modal dialog box that shows
; "Copyright 1987 by Leo Scanlon" and an OK button. If the user chooses
; this item again, the box shows "As I said before:" in addition to the
; original text.

        absaddr on
        MCOPY Modald.macros

Modald      START
            using GlobalData

; -----
;
; Global equates used throughout the program.

ScreenMode    gequ $80                ;640 mode, no fill
MaxX          gequ 640                ;640 mode for Event Manager

            phk                        ;Set data bank to program
            plb                        ; bank to allow absolute addressing

            jsr InitStuff              ;Initialize everything
            bcs AllDone               ;Quit if initialization fails

            stz QuitFlag               ;Initialize the quit flag to 0
            jsr EventLoop              ;Let the user play

AllDone      anop                     ;All is done, shut down
            PushLong Win1Ptr           ;Close the windows
            _CloseWindow
            PushLong Win2Ptr
            _CloseWindow
            PushLong Win3Ptr
            _CloseWindow

            _DeskShutDown              ;Desk Manager
            _MenuShutDown             ;Menu Manager
            _WindShutDown              ;Window Manager
            _DialogShutDown            ;Dialog Manager
            _CtlShutDown               ;Control Manager
            _EMShutDown                ;Event Manager
            _LEShutDown                ;LineEdit
            _QDShutDown                ;QuickDraw II
            _MTShutDown                ;Miscellaneous Tools
```


380 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

```
; Item types for dialog box.
```

```
ButtonItem    equ 10
StatText      equ 15
```

```
; To disable an item type, add ItemDisable to it.
```

```
ItemDisable   equ $8000
```

```
END
```

```
;*****
```

```
;
```

```
; InitStuff
```

```
;
```

```
; Initializes tool sets, gets space in bank 0 for use as direct  
; page by tool sets that need it, and ensures that RAM-based  
; tools are in memory.
```

```
;
```

```
;*****
```

```
InitStuff      START
                using GlobalData
                using MenuData
```

```
; Initialize the Tool Locator, Memory Manager, and Miscellaneous  
; Tools.
```

```
    _TLStartup          ;Tool locator

    PushWord #0          ;Memory Manager
    _MMStartup

    pla                 ;Memory Manager returns program's ID
    sta MyID

    _MTStartup          ;Misc. Tools
    ldx #3
    jsr PrepareToDie
```

```
; Get some space for the direct page we need. QuickDraw needs  
; three pages and each Manager except Window needs one page.
```

```
    PushLong #0          ;Space for handle
    PushLong #$700       ;Seven pages
    PushWord MyID        ;Owner
    PushWord #$C001      ;Locked, fixed, fixed bank
    PushLong #0          ;Location
    _NewHandle
    ldx #$FF
    jsr PrepareToDie

    pla                 ;Read handle and store in dp
    sta 0
    pla
    sta 2
```

```

        lda [0]                ;Get dp location from handle
        sta 4                  ; and store at loc 4 of dp

; Initialize QuickDraw

        pha                    ;Use dp obtained from handle
        PushWord #ScreenMode   ;Mode = 640
        PushWord #160          ;Max size of scan line (in bytes)
        PushWord MyID
        _QDStartup
        ldx #4
        jsr PrepareToDie

; Initialize Event Manager

        lda 4                  ;dp to use = QD dp + $300
        clc
        adc #$300
        sta 4
        pha
        PushWord #20           ;Queue size
        PushWord #0            ;X clamp left
        PushWord #MaxX         ;X clamp right
        PushWord #0            ;Y clamp top
        PushWord #200          ;Y clamp bottom
        PushWord MyID
        _EMStartup
        ldx #6
        jsr PrepareToDie

;-----
;
; Load the RAM-based tools I need.

LoadAgain    PushLong #ToolTable
              _LoadTools
              bcc ToolsLoaded

              cmp #VolNotFound
              beq DoMount
              sec
              ldx #$FE
              jsr PrepareToDie

DoMount      anop
              jsr MountBootDisk
              cmp #1
              beq LoadAgain

              sec
              rts

; The tools have been loaded. Initialize the Window Manager, Control
; Manager, LineEdit, Dialog Manager, Menu Manager, and Desk Manager.

ToolsLoaded  anop
              PushWord MyID          ;Window Manager
              _WindStartup
              ldx #14
              jsr PrepareToDie

```

```

        PushLong #0                ;Prepare screen for windows
        _RefreshDesktop

        PushWord MyID              ;Control Manager
        lda 4                      ;dp to use = EM dp + $100
        clc
        adc #$100
        sta 4
        pha
        _CtlStartup
        ldx #16
        jsr PrepareToDie

        PushWord MyID              ;LineEdit
        lda 4
        clc
        adc #$100
        sta 4
        pha
        _LEStartup
        ldx #20
        jsr PrepareToDie

        PushWord MyID              ;Dialog Manager
        _DialogStartup
        ldx #21
        jsr PrepareToDie

        PushWord MyID              ;Menu Manager
        lda 4
        clc
        adc #$100
        pha
        _MenuStartup
        ldx #15
        jsr PrepareToDie

        _DeskStartup              ;Desk Manager

;-----
;
; Build the menu bar by inserting the three menus (right to left order).

        PushLong #0                ;Windows menu
        PushLong #WindowsMenu
        _NewMenu
        PushWord #0
        _InsertMenu

        PushLong #0                ;File menu
        PushLong #FileMenu
        _NewMenu
        PushWord #0
        _InsertMenu

        PushLong #0                ;Apple menu
        PushLong #AppleMenu
        _NewMenu
        PushWord #0
        _InsertMenu

```

```

;-----
;
; Call the Desk Manager to install the list of NDAs in the system.

        PushWord #1                ;NDAs will go in Apple menu
        _FixAppleMenu

;-----
;
; Finish off getting the menu bar ready.

        PushWord #0
        _FixMenuBar
        pla                        ;Discard menu bar height

        _DrawMenuBar              ;Display the completed bar

        jsr SetUpWindows          ;Show the windows
        _ShowCursor              ; and the mouse pointer

        jsr DrawWindows           ;Draw the window contents

        clc                      ;Clear the carry flag
        rts                      ; and return

        END

;*****
;
; Set Up Windows
;
; Opens the three windows (but does not draw their contents).
;
;*****

SetUpWindows START
        using GlobalData
        using WindowData

        PushLong #0               ;Window 1
        PushLong #Win1ParamBlock
        _NewWindow

        pla
        sta Win1Ptr
        pla
        sta Win1Ptr+2

        PushLong #0               ;Window 2
        PushLong #Win2ParamBlock
        _NewWindow

        pla
        sta Win2Ptr
        pla
        sta Win2Ptr+2

        PushLong #0               ;Window 3
        PushLong #Win3ParamBlock
        _NewWindow

```

384 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

```

        pla
        sta Win3Ptr
        pla
        sta Win3Ptr+2

        rts
        END

;*****
;
; Event Loop
;
; Display windows until user chooses "Quit".
;
;*****

EventLoop      START
                using GlobalData

Again          lda QuitFlag
                bne NoMore

                PushWord #0                ;Space for result
                PushWord #$FFFF            ;Accept any event
                PushLong #TaskRecord        ;Point to task record buffer
                _TaskMaster

                pla                        ;Is an event available?
                beq Again                  ; No. Continue polling
                asl a                      ; Yes. Double it
                tax                        ; and copy it into X.

                jsr (TaskTable,x)          ;Execute the event's routine,
                bra Again                  ; then resume polling

NoMore         rts

TaskTable      anop                      ;Event Manager events
                dc i'Ignore'              ; 0 null
                dc i'Ignore'              ; 1 mouse down
                dc i'Ignore'              ; 2 mouse up
                dc i'Ignore'              ; 3 key down
                dc i'Ignore'              ; 4 undefined
                dc i'Ignore'              ; 5 auto-key
                dc i'Ignore'              ; 6 update
                dc i'Ignore'              ; 7 undefined
                dc i'Ignore'              ; 8 activate
                dc i'Ignore'              ; 9 switch
                dc i'Ignore'              ; 10 desk acc
                dc i'Ignore'              ; 11 device driver
                dc i'Ignore'              ; 12 ap
                dc i'Ignore'              ; 13 ap
                dc i'Ignore'              ; 14 ap
                dc i'Ignore'              ; 15 ap
                dc i'Ignore'              ; 16 in desktop
                dc i'DoMenu'              ; 17 in system menu bar
                dc i'Ignore'              ; 18 in system window
                dc i'Ignore'              ; 19 in window content region
                dc i'Ignore'              ; 20 in drag region (title bar)
                dc i'Ignore'              ; 21 in grow box

```



```

        dc i'DoClose'          ; 22 in go-away region (close box)
        dc i'Ignore'           ; 23 in zoom box
        dc i'Ignore'           ; 24 in information bar
        dc i'Ignore'           ; 25 in right scroll bar
        dc i'Ignore'           ; 26 in bottom scroll bar
        dc i'Ignore'           ; 27 in frame
        dc i'Ignore'           ; 28 in drop region
    END

;*****
;
; Ignore
;
; This do-nothing subroutine returns to the event loop for
; events we don't want.
;
;*****

Ignore    START
          rts
          END

;*****
;
; DoMenu
;
; Called when TaskMaster tells me that a menu item has been
; selected.
;
;*****

DoMenu    START
          using GlobalData

          lda TaskData          ;Get the item ID
          and #$00FF            ; and strip off the "256" bit
          asl a                 ;Double the result
          tax                   ; and copy it to X

          jsr (ItemTable,x)     ;Call the item's subroutine,

          PushWord #0           ; then unhighlight the menu
          PushWord TaskData+2
          _HiliteMenu

          rts

ItemTable dc i'DoAbout'        ;From Apple menu
          dc i'DoQuit'         ;From File menu
          dc i'DoWin1'         ;From Windows menu
          dc i'DoWin2'
          dc i'DoWin3'

          END

;*****
;
; Do Quit
;
; Sets the quit flag.
;
;*****

```

386 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

```

DoQuit          START
                using GlobalData

                lda #1
                sta QuitFlag
                rts

                END

;*****
;
; Do About
;
; Brings up About box and waits until user presses OK button
; before putting it away.
;
;*****

DoAbout         START
                using GlobalData
                using WindowData

                PushLong #0                ;Space for result (pointer)
                PushLong #DRect            ;Pointer to content rect.
                PushWord #1                ;Make window visible
                PushLong #0                ;No particular value
                _NewModalDialog

                pla                        ;Store pointer to port
                sta MDialogPtr
                pla
                sta MDialogPtr+2

                PushLong MDialogPtr        ;Set up the OK button
                PushWord #1                ;Item #1 (default)
                PushLong #OKBtnRect        ;Display rectangle
                PushWord #ButtonItem        ;OK is a simple button
                PushLong #ButtonText        ;Pointer to its text
                PushWord #0                ;Initial value
                PushWord #0                ;Default flag
                PushLong #0                ;Default color table
                _NewItem

                lda NotFirstTime            ;First time for this box?
                beq ShowText2

                PushLong MDialogPtr        ; No. Display "As I said before:"
                PushWord #2                ;Item #2
                PushLong #TextRect1
                PushWord #StatText+ItemDisable
                PushLong #Text1
                PushWord #0
                PushWord #0
                PushLong #0
                _NewItem

ShowText2       PushLong MDialogPtr        ;Display copyright message
                PushWord #3                ;Item #3
                PushLong #TextRect2
                PushWord #StatText+ItemDisable
                PushLong #Text2
                PushWord #0

```

```

        PushWord #0
        PushLong #0
        _NewItem

        PushWord #0           ;Box only has one selectable item
        PushLong #0
        _ModalDialog

        pla                   ;I know which item it was. Ignore it.

        PushLong MDialogPtr   ;Take the box off the screen
        _CloseDialog

        lda #1                ;From here on, display message 1
        sta NotFirstTime
        rts

END

;*****
;
; DoWin1
;
; Selects and shows window 1 in response to menu selection.
;
;*****

DoWin1      START
            using GlobalData

            PushLong Win1Ptr
            _SelectWindow

            PushLong Win1Ptr
            _ShowWindow

            rts
            END

;*****
;
; DoWin2
;
; Selects and shows window 2 in response to menu selection.
;
;*****

DoWin2      START
            using GlobalData

            PushLong Win2Ptr
            _SelectWindow

            PushLong Win2Ptr
            _ShowWindow

            rts
            END

```

388 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

```
;*****
;
; DoWin3
;
; Selects and shows window 3 in response to menu selection.
;
;*****

DoWin3          START
                using GlobalData

                PushLong Win3Ptr
                _SelectWindow

                PushLong Win3Ptr
                _ShowWindow

                rts
                END

;*****
;
; DoClose
;
; Hides the active window when user presses button in its
; close box.
;
;*****

DoClose         START
                using GlobalData

                PushLong TaskData
                _HideWindow

                rts
                END

;*****
;
; Menu Data
;
;*****

MenuData        DATA

AppleMenu       dc c'>L@N1X',i1'13'
                dc c' LAbout this program...\N256',i1'13'
                dc c'.'

FileMenu        dc c'>L File \N2',i1'13'
                dc c' LQuit\N257*Qq',i1'13' ;Key command = Apple-Q
                dc c'.'

WindowsMenu     dc c'>L Windows \N3',i1'13'
                dc c' LWindow 1\N258',i1'13'
                dc c' LWindow 2\N259',i1'13'
                dc c' LWindow 3\N260',i1'13'
                dc c'.'

                END
```

```

;*****
;
; Window Data
;
;*****

WindowData      DATA

Win1ParamBlock anop
    dc    i'Win1End-Win1ParamBlock'
    dc    i2'%1101110110000000' Everything but info bar
    dc    i4'Win1Title'         Pointer to window's title
    dc    i4'0'                 Reserved (wRefCon)
    dc    i2'26,0,190,620'      Zoomed content region
    dc    i4'0'                 Default color table
    dc    i2'0'                 Vertical origin
    dc    i2'0'                 Horizontal origin
    dc    i2'200'               Data area height
    dc    i2'640'               Data area width
    dc    i2'200'               Max grow height
    dc    i2'640'               Max grow width
    dc    i2'4'                 Number of pixels to scroll vertically.
    dc    i2'16'               Number of pixels to scroll
horizontally.
    dc    i2'40'               Number of pixels to page vertically.
    dc    i2'160'              Number of pixels to page horizontally.
    dc    i4'0'                 Information bar text string.
    dc    i2'0'                 Info bar height
    dc    i4'0'                 Routine to draw shape (none, standard)
    dc    i4'0'                 Routine to draw info. bar.
    dc    i4'DrawWin1'          Routine to draw content.
    dc    i'40,20,100,360'      Size/pos of content
    dc    i4'-1'                Window's order (-1 means topmost)
    dc    i4'0'                 Window Manager allocates wind. record
Win1End      anop

Win2ParamBlock anop
    dc    i'Win2End-Win2ParamBlock'
    dc    i2'%1101110110000000' Everything but info bar
    dc    i4'Win2Title'         Pointer to window's title
    dc    i4'0'                 Reserved (wRefCon)
    dc    i2'26,0,190,620'      Zoomed content region
    dc    i4'0'                 Default color table
    dc    i2'0'                 Vertical origin
    dc    i2'0'                 Horizontal origin
    dc    i2'200'               Data area height
    dc    i2'640'               Data area width
    dc    i2'200'               Max grow height
    dc    i2'640'               Max grow width
    dc    i2'4'                 Number of pixels to scroll vertically.
    dc    i2'16'               Number of pixels to scroll
horizontally.
    dc    i2'40'               Number of pixels to page vertically.
    dc    i2'160'              Number of pixels to page horizontally.
    dc    i4'0'                 Information bar text string.
    dc    i2'0'                 Info bar height
    dc    i4'0'                 Routine to draw shape (none, standard)
    dc    i4'0'                 Routine to draw info. bar.
    dc    i4'DrawWin2'          Routine to draw content.
    dc    i'50,30,110,380'      Size/pos of content
    dc    i4'-1'                Window's order (-1 means topmost)
    dc    i4'0'                 Window Manager allocates wind. record
Win2End      anop

```

390 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

```

Win3ParamBlock anop
    dc    i'Win3End-Win3ParamBlock'
    dc    i2'*1101110110000000'    Everything but info bar
    dc    i4'Win3Title'              Pointer to window's title
    dc    i4'0'                      Reserved (wRefCon)
    dc    i2'26,0,190,620'           Zoomed content region
    dc    i4'0'                      Default color table
    dc    i2'0'                      Vertical origin
    dc    i2'0'                      Horizontal origin
    dc    i2'200'                    Data area height
    dc    i2'640'                    Data area width
    dc    i2'200'                    Max grow height
    dc    i2'640'                    Max grow width
    dc    i2'4'                      Number of pixels to scroll vertically.
    dc    i2'16'                    Number of pixels to scroll
horizontally.
    dc    i2'40'                    Number of pixels to page vertically.
    dc    i2'160'                   Number of pixels to page horizontally.
    dc    i4'0'                    Info bar text string.
    dc    i2'0'                    Info bar height
    dc    i4'0'                    Routine to draw shape (none, standard)
    dc    i4'0'                    Routine to draw info. bar.
    dc    i4'DrawWin3'             Routine to draw content.
    dc    i'60,40,120,400'         Size/pos of content
    dc    i4'-1'                   Window's order (-1 means topmost)
    dc    i4'0'                    Window Manager allocates wind. record
Win3End    anop

Win1Title    str 'Window 1'
Win2Title    str 'Window 2'
Win3Title    str 'Window 3'

PenPat1      dc h'11 11 11 11 11 11 11 11' ;Dithered blue pen pattern
              dc h'11 11 11 11 11 11 11 11'
              dc h'11 11 11 11 11 11 11 11'
              dc h'11 11 11 11 11 11 11 11'

Rect          dc i'20,20,40,40'      ;Rectangle painted in windows

; The rest of this data is for the dialog box.

MDialogPtr    ds 4

DRect          dc i'35,150,135,490'  ;BoundsRect of dialog window
OKBtnRect      dc i'85,280,0,0'      ;OK button display rect

TextRect1      dc i'15,10,35,500'
Text1          str 'As I said before:'

TextRect2      dc i'35,10,55,500'
Text2          str 'Copyright 1987 by Leo Scanlon'

ButtonText     str 'OK'

```

END

```

;*****
;
; Draw Windows
;
; Fills in the content region of each window, working rearward.
;
;*****

```

```
DrawWindows  START
              using GlobalData
```

```
    PushLong Win3Ptr          ;Window 3
    _SetPort
```

```
    jsl DrawWin3
```

```
    PushLong Win3Ptr
    _ShowWindow
```

```
    PushLong Win2Ptr          ;Window 2
    _SetPort
```

```
    jsl DrawWin2
```

```
    PushLong Win2Ptr
    _ShowWindow
```

```
    PushLong Win1Ptr          ;Window 1
    _SetPort
```

```
    jsl DrawWin1
```

```
    PushLong Win1Ptr
    _ShowWindow
```

```
    rts
    END
```

```
*****
;
; Draw Window 1
;
; Draws the contents of Window 1--a blue box on a white
; background. Called by DrawWindows initially and by the Window
; Manager when it updates the window.
;
*****
```

```
DrawWin1  START
          using WindowData
```

```
    PushWord #3                ;Background is white
    _SetSolidBackPat
```

```
    PushLong #PenPat1          ;Set pen pattern to blue
    _SetPenPat
```

```
    PushLong #Rect             ; and paint the rectangle
    _PaintRect
```

```
    rtl                        ;RTL required for update events
    END
```

392 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

```
;*****
;
; Draw Window 2
;
; Draws the contents of Window 2--a white box on a green
; background. Called by DrawWindows initially and by the Window
; Manager when it updates the window.
;
;*****
```

```
DrawWin2 START
    using WindowData

    PushWord #2                ;Background is green
    _SetSolidBackPat

    PushWord #3                ;Set pen pattern to white
    _SetSolidPenPat

    PushLong #Rect             ; and paint the rectangle
    _PaintRect

    rtl                        ;RTL required for update events
    END
```

```
;*****
;
; Draw Window 3
;
; Draws the contents of Window 3--a green box on a red
; background. Called by DrawWindows initially and by the Window
; Manager when it updates the window.
;
;*****
```

```
DrawWin3 START
    using WindowData

    PushWord #1
    _SetSolidBackPat

    PushWord #2                ;Set pen pattern to green
    _SetSolidPenPat

    PushLong #Rect             ; and paint the rectangle
    _PaintRect

    rtl                        ;RTL required for update events
    END
```

```
*****
*
* Prepare To Die
*
* Dies if carry is set.
*
* Error number to display is in X register.
* Assumes that Miscellaneous Tools is active.
*
*****
```



```

PrepareToDie  START
               bcs RealDeath      ;Carry = 1?
               rts                 ; No. Return to caller

RealDeath     phx                 ; Yes. Goodbye, program.
               PushLong #DeathMsg
               _SysFailMgr

DeathMsg      str 'Could not handle error '

               END

;*****
;
; Mount Boot Disk
;
; Tells the user to insert the disk that contains the RAM-based
; tools.
;
;*****

MountBootDisk START

               _Set_Prefix SetPrefixParams
               _Get_Prefix GetPrefixParams

               PushWord #0          ;Space for result
               PushWord #195        ;Column position for dialog box
               PushWord #30         ;Row position for dialog box
               PushLong #PromptStr  ;Prompt at top of dialog box
               PushLong #VolStr     ;Volume name string
               PushLong #OKStr      ;String in Button 1
               PushLong #CancelStr  ;String in Button 2
               _TLMountVolume

               pla                  ;Obtain the button number
               rts                 ; and return to caller

PromptStr     str 'Please insert the disk.'
VolStr        ds 16
OKStr         str 'OK'
CancelStr     str 'Shutdown'

GetPrefixParams dc i'7'
               dc i4'VolStr'

SetPrefixParams dc i'7'
               dc i4'BootStr'

BootStr       str '*/'

               END

```

The box is quite simple; it contains only text and an OK button. The text can take either of two forms, however. The first time the user displays the box, it contains "Copyright 1987 by Leo Scanlon." After that, whenever

the user accesses the box, it shows not only the copyright notice, but also the clause “As I said before.”; this serves to illustrate how you can actually change dialog windows between appearances.

MODALD isn’t very different from MENUS, actually. Its GlobalData segment contains a NotFirstTime parameter that regulates what appears in the box. It’s set to 0 at the beginning of the program, then changed when the box first appears. GlobalData also contains the item types for the dialog (ButtonItem and StatText, in this case) and an ItemDisable that lets me disable the text.

The InitStuff segment contains start-up calls for the Line Editor and the Dialog Manager. Their shutdown calls are in the first segment, ModalD.) The item table in the DoMenu segment now includes the address of a DoAbout subroutine that brings up the dialog box and keeps it on the screen until the user presses OK.

DoAbout has what you might expect. It starts with a NewModalDialog call to create the box and NewDItem calls for the three items in it — a button and two lines of static text. (The data for these items is in the WindowData segment.) DoAbout ends with a ModalDialog call, which waits for the user to press OK, and a CloseDialog call to remove the box from the screen.

Tool Calls for Alert Boxes

Alert boxes are easiest of all to program, because they require only a single tool call. And that call does all the work of creating the box, interacting with the user, and closing the box. There are four calls that do these jobs: Alert, StopAlert, NoteAlert, and CautionAlert. As Table 12-4 shows, these calls differ only in what they display at the top left-hand corner. Alert displays nothing, while StopAlert, NoteAlert, and CautionAlert display an icon; a hand, a “talking” face, or an exclamation point, respectively (see Figure 12-6).



Stop



Note



Caution

Figure 12-6

Table 12-4

<u>Alert</u>		Draw general-purpose alert box
Call with:	PushWord #0 ;Space for result	
	PushLong <i>AlertTemplate</i> ;Pointer to an alert template	
	PushLong <i>filterProc</i> ;Pointer to filter used by ModalDialog	
	<u>Alert</u>	
Result:	ID of item hit (word).	
Note:	See text for description of the alert template.	
<u>StopAlert</u>		Draw Stop alert box
Call with:	PushWord #0 ;Space for result	
	PushLong <i>AlertTemplate</i> ;Pointer to an alert template	
	PushLong <i>filterProc</i> ;Pointer to filter used by ModalDialog	
	<u>StopAlert</u>	
Result:	ID of item hit (word).	
Note:	Same as Alert, but draws the Stop icon somewhere near the top left corner of the box.	
<u>NoteAlert</u>		Draw Note alert box
Call with:	PushWord #0 ;Space for result	
	PushLong <i>AlertTemplate</i> ;Pointer to an alert template	
	PushLong <i>filterProc</i> ;Pointer to filter used by ModalDialog	
	<u>NoteAlert</u>	
Result:	ID of item hit (word).	
Note:	Same as Alert, but draws the Note icon somewhere near the top left corner of the box.	
<u>CautionAlert</u>		Draw Caution alert box
Call with:	PushWord #0 ;Space for result	
	PushLong <i>AlertTemplate</i> ;Pointer to an alert template	
	PushLong <i>filterProc</i> ;Pointer to filter used by ModalDialog	
	<u>CautionAlert</u>	
Result:	ID of item hit (word).	
Note:	Same as Alert, but draws the Caution icon somewhere near the top left corner of the box.	

```

dc i1 '%10000010' ;Stage3
; (two beeps)
dc i1 '%10000011' ;Stage4
; (three beeps)
dc i4 'ButtonTemp' ; Pointer to
; button template
dc i4 'TextTemp' ;Pointer to text
; template
dc i4 '0' ;Terminator

```

Item Template

The item templates pointed to here are simply the inputs you use to set up dialog items with `NewDItem`, but in “template” form — that is, `itemID` (word), `itemRect` (rect), `itemType` (word), `itemDescr` (log), `itemValue` (word), `itemFlag` (word), and `itemColor` (long). For example, the `ButtonTemp` template mentioned earlier might look like this:

```
ButtonTemp dc i'1'           ;ID
           dc i'85,200,0,0'   ;Display rect
           dc i'ButtonItem'   ;Simple button
           dc i4'ButtonText' ;Pointer to its text
           dc i'0'           ;Initial value
           dc i'0'           ;Default flag
           dc i4'0'          ;Default color table
```

Because everything is tucked neatly away in templates, the subroutine that displays the alert box can be quite short. If you have only a button enabled, for example (a typical situation with alert boxes), the subroutine might be:

```
PushWord #0           ;Space for result
PushLong #AlertTemp   ;Pointer to alert template
PushLong #0           ;No particular value
_StopAlert

pla                   ;It could only be one item.
rts                   ;Ignore it and leave.
```

Final Comments

Within these pages, I introduced you to the workings of the 65816 microprocessor and its role in the Apple IIGS. I also covered the Apple IIGS Programmer’s Workshop for assembly language and demonstrated how you can use tool sets within the IIGS Toolbox to do some common programming operations.

Where you go from here is, of course, up to you. It depends on what you want your programs to *do*. To create and play music, you need the services of the Sound Manager. To produce fancy, Macintosh-style lettering, you need the Font Manager. And so on. The point is that the Toolbox provides

tool sets and tools to do just about anything you want, and they're all well documented in the *Apple IIGS Toolbox Reference*. Moreover, if you can't find a tool set that meets some specific need, you can write one of your own! Details about doing that are also in the Toolbox Reference.

Well, all books must end somewhere, and this one ends here. I have genuinely enjoyed this journey through the resources of the Apple IIGS, and hope you have, too.

APPENDIX A

Hexadecimal/Decimal Conversion

HEXADECIMAL COLUMNS											
6		5		4		3		2		1	
HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC
0	0	0	0	0	0	0	0	0	0	0	0
1	1,048,576	1	65,536	1	4,096	1	256	1	16	1	1
2	2,097,152	2	131,072	2	8,192	2	512	2	32	2	2
3	3,145,728	3	196,608	3	12,288	3	768	3	48	3	3
4	4,194,304	4	262,144	4	16,384	4	1,024	4	64	4	4
5	5,242,880	5	327,680	5	20,480	5	1,280	5	80	5	5
6	6,291,456	6	393,216	6	24,576	6	1,536	6	96	6	6
7	7,340,032	7	458,752	7	28,672	7	1,792	7	112	7	7
8	8,388,608	8	524,288	8	32,768	8	2,048	8	128	8	8
9	9,437,184	9	589,824	9	36,864	9	2,304	9	144	9	9
A	10,485,760	A	655,360	A	40,960	A	2,560	A	160	A	10
B	11,534,336	B	720,896	B	45,056	B	2,816	B	176	B	11
C	12,582,912	C	786,432	C	49,152	C	3,072	C	192	C	12
D	13,631,488	D	851,968	D	53,248	D	3,328	D	208	D	13
E	14,680,064	E	917,504	E	57,344	E	3,584	E	224	E	14
F	15,728,640	F	983,040	F	61,440	F	3,840	F	240	F	15
7654		3210		7654		3210		7654		3210	
Byte				Byte				Byte			

POWERS OF 2

2^n	n
256	8
512	9
1,024	10
2,048	11
4,096	12
8,192	13
16,384	14
32,768	15
65,536	16
131,072	17
262,144	18
524,288	19
1,048,576	20
2,097,152	21
4,194,304	22
8,388,608	23
16,777,216	24

$2^0 = 16^0$
$2^4 = 16^1$
$2^8 = 16^2$
$2^{12} = 16^3$
$2^{16} = 16^4$
$2^{20} = 16^5$
$2^{24} = 16^6$
$2^{28} = 16^7$
$2^{32} = 16^8$
$2^{36} = 16^9$
$2^{40} = 16^{10}$
$2^{44} = 16^{11}$
$2^{48} = 16^{12}$
$2^{52} = 16^{13}$
$2^{56} = 16^{14}$
$2^{60} = 16^{15}$

POWERS OF 16

16^n	n
1	0
16	1
256	2
4,096	3
65,536	4
1,048,576	5
16,777,216	6
268,435,456	7
4,294,967,296	8
68,719,476,736	9
1,099,511,627,776	10
17,592,186,044,416	11
281,474,976,710,656	12
4,503,599,627,370,496	13
72,057,594,037,927,936	14
1,152,921,504,606,846,976	15

APPENDIX B

ASCII Table

MSD		0	1	2	3	4	5	6	7
LSD		000	001	010	011	100	101	110	111
0	0000	NUL	DLE	SP	0	@	P		p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	"	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	\$	4	D	T	d	t
5	0101	ENQ	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	'	7	G	W	g	w
8	1000	BS	CAN	(8	H	X	h	x
9	1001	HT	EM)	9	I	Y	i	y
A	1010	LF	SUB	*	:	J	Z	j	z
B	1011	VT	ESC	+	;	K	[k	{
C	1100	FF	FS	,	<	L	\	l	
D	1101	CR	GS	-	=	M]	m	}
E	1110	SO	RS	.	>	N	^	n	~
F	1111	SI	US	/	?	O	_	o	DEL

NUL — Null
 SOH — Start of Heading
 STX — Start of Text
 ETX — End of Text
 EOT — End of Transmission
 ENQ — Enquiry
 ACK — Acknowledge
 BEL — Bell
 BS — Backspace
 HT — Horizontal Tabulation
 LF — Line Feed
 VT — Vertical Tabulation
 FF — Form Feed
 CR — Carriage Return
 SO — Shift Out
 SI — Shift In

DLE — Data Link Escape
 DC — Device Control
 NAK — Negative Acknowledge
 SYN — Synchronous Idle
 ETB — End of Transmission Block
 CAN — Cancel
 EM — End of Medium
 SUB — Substitute
 ESC — Escape
 FS — File Separator
 GS — Group Separator
 RS — Record Separator
 US — Unit Separator
 SP — Space (Blank)
 DEL — Delete

65816 Instruction Set Summary

This appendix summarizes the 65816 instruction set in alphabetical order. It includes a short description of what each instruction does and which status flags it affects. Each entry also contains a table that lists the allowable addressing modes, the assembler format, the opcode, the number of bytes the instruction occupies in memory, and how long it takes to execute. The Assembler Format column includes the following abbreviations:

Num = 8- or 16-bit constant

Loc = 16-bit address

LongLoc = 24-bit address

DLoc = 16-bit address in the direct page

DLong = 24-bit address in the direct page

Dis8 = 8-bit signed relative displacement (distance forward or backward)

Dis16 = 16-bit signed relative displacement

Execution times are shown in clock cycles. To convert them to nanoseconds, multiply by 400 for a 2.5-MHz clock or by 1,000 for a 1-MHz clock. (Since 1,000 nanoseconds is 1 microsecond, you get microseconds if you use the cycle count directly.) The execution times are convenient for comparing the efficiency of one instruction or addressing mode with another. This helps you decide which approach is best, in case you have doubts.

ADC*Add to Accumulator with Carry*

Operation: Add the operand and the carry flag to the accumulator.

N	V	M	X	D	I	Z	C
*	*	*	*

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Immediate	ADC #Num	69	3 ²	3 ³
Absolute	ADC Loc	6D	3	5 ³
Absolute long	ADC LongLoc	6F	4	6 ³
Absolute indexed with X	ADC Loc,X	7D	3	5 ^{1,3}
Absolute long indexed with X	ADC LongLoc,X	7F	4	6 ³
Absolute indexed with Y	ADC Loc,Y	79	3	5 ^{1,3}
Direct	ADC DLoc	65	2	4 ^{3,4}
Direct indexed with X	ADC DLoc,X	75	2	5 ^{3,4}
Direct indirect	ADC (DLoc)	72	2	6 ^{3,4}
Direct indirect long	ADC [DLong]	67	2	7 ^{3,4}
Direct indirect indexed	ADC (DLoc),Y	71	2	6 ^{1,3,4}
Direct indirect indexed long	ADC [DLong],Y	77	2	7 ^{3,4}
Direct indexed indirect	ADC (DLoc,X)	61	2	7 ^{3,4}
Stack relative	ADC Dis8,S	63	2	5 ³
Stack relative indirect indexed	ADC (Dis8,S),Y	73	2	8 ³

¹Add 1 cycle if adding index crosses a page boundary.²Subtract 1 byte if M = 1 (8-bit memory/accumulator).³Subtract 1 cycle if M = 1 (8-bit memory/accumulator).⁴Add 1 cycle if direct register low (DL) is not equal to 0.

AND*AND Memory with Accumulator*

Operation: ANDs the operand with the accumulator.

N V M X D I Z C
* *

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Immediate	AND #Num	29	3 ²	3 ³
Absolute	AND Loc	2D	3	5 ³
Absolute long	AND LongLoc	2F	4	6 ³
Absolute indexed with X	AND Loc,X	3D	3	5 ^{1,3}
Absolute long indexed with X	AND LongLoc,X	3F	4	6 ³
Absolute indexed with Y	AND Loc,Y	39	3	5 ^{1,3}
Direct	AND DLoc	25	2	4 ^{3,4}
Direct indexed with X	AND DLoc,X	35	2	5 ^{3,4}
Direct indirect	AND (DLoc)	32	2	6 ^{3,4}
Direct indirect long	AND [DLong]	27	2	7 ^{3,4}
Direct indirect indexed	AND (DLoc),Y	31	2	6 ^{1,3,4}
Direct indirect indexed long	AND [DLong],Y	37	2	7 ^{3,4}
Direct indexed indirect	AND (DLoc,X)	21	2	7 ^{3,4}
Stack relative	AND Dis8,S	23	2	5 ³
Stack relative indirect indexed	AND (Dis8,S),Y	33	2	8 ³

¹Add 1 cycle if adding index crosses a page boundary.

²Subtract 1 byte if M = 1 (8-bit memory/accumulator).

³Subtract 1 cycle if M = 1 (8-bit memory/accumulator).

⁴Add 1 cycle if direct register low (DL) is not equal to 0.

ASL*Arithmetic Shift Left*

Operation: Shifts the operand left one bit position. It puts the displaced bit in the carry flag (C) and puts a 0 in the vacated bit position. See also LSR, which shifts operands right.

N V M X D I Z C
* *

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Accumulator	ASL A	0A	1	2
Absolute	ASL Loc	0E	3	8 ²
Absolute indexed with X	ASL Loc,X	1E	3	9 ^{1,2}
Direct	ASL DLoc	06	2	7 ^{2,3}
Direct indexed with X	ASL DLoc,X	16	2	8 ^{2,3}

¹Add 1 cycle if adding index crosses a page boundary.

²Subtract 2 cycles if M = 1 (8-bit memory/accumulator).

³Add 1 cycle if direct register low (DL) is not equal to 0.

BCC
BLT*Branch on Carry Clear*
Branch if Less Than

Operation: Branches on C = 0.

N	V	M	X	D	I	Z	C
.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Program counter relative	BCC Dis8	90	2	2 ¹

¹Add 1 cycle if branch is taken. In emulation mode only, add 1 additional cycle if the branch taken is to a different page.

BCS
BGE*Branch on Carry Set*
Branch if Greater Than or Equal

Operation: Branches on C = 1.

N	V	M	X	D	I	Z	C
.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Program counter relative	BCS Dis8	B0	2	2 ¹

¹Add 1 cycle if branch is taken. In emulation mode only, add 1 additional cycle if the branch taken is a different page.

BEQ*Branch if Equal*

Operation: Branches on Z = 1.

N	V	M	X	D	I	Z	C
.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Program counter relative	BEQ Dis8	F0	2	2 ¹

¹Add 1 cycle if branch is taken. In emulation mode only, add 1 additional cycle if the branch taken is to a different page.

BGE — See **BCS**

BIT*Bit Test*

Operation: ANDs the operand with the accumulator, but reports the result only in the flags.
The operands are unaffected.

	N	V	M	X	D	I	Z	C
Immediate mode	*	.
Other modes, 8-bit	M ⁷	M ⁶	*	.
Other modes, 16-bit	M ¹⁵	M ¹⁴	*	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Immediate	BIT #Num	89	3 ²	3 ³
Absolute	BIT Loc	2C	3	5 ³
Absolute indexed with X	BIT Loc,X	3C	3	5 ^{1,3}
Direct	BIT DLoc	24	2	4 ^{3,4}
Direct indexed with X	BIT DLoc,X	34	2	5 ^{3,4}

¹Add 1 cycle if adding index crosses a page boundary.

²Subtract 1 byte if M = 1 (8-bit memory/accumulator).

³Subtract 1 cycle if M = 1 (8-bit memory/accumulator).

⁴Add 1 cycle if direct register low (DL) is not equal to 0.

BLT — See **BCC**

BMI*Branch if Minus*

Operation: Branches on N = 1.

N	V	M	X	D	I	Z	C
.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Program counter relative	BMI Dis8	30	2	2 ¹

¹Add 1 cycle if branch is taken. In emulation mode only, add 1 additional cycle if the branch taken is to a different page.

BNE*Branch if Not Equal*

Operation: Branches on Z = 0.

N	V	M	X	D	I	Z	C
.

Addressing Mode	Assembler Format		Opcode	Bytes	Cycles
Program counter relative	BNE	Dis8	D0	2	2 ¹

¹Add 1 cycle if branch is taken. In emulation mode only, add 1 additional cycle if the branch taken is to a different page.

BPL*Branch if Plus*

Operation: Branches on N = 0.

N	V	M	X	D	I	Z	C
.

Addressing Mode	Assembler Format		Opcode	Bytes	Cycles
Program counter relative	BPL	Dis8	10	2	2 ¹

¹Add 1 cycle if branch is taken. In emulation mode only, add 1 additional cycle if the branch taken is a different page.

BRA*Branch Always*Operation: Branches unconditionally. See also **BRL**.

N	V	M	X	D	I	Z	C
.

Addressing Mode	Assembler Format		Opcode	Bytes	Cycles
Program counter relative	BRA	Dis8	80	2	2 ¹

¹Add 1 cycle if branch is taken. In emulation mode only, add 1 additional cycle if the branch taken is to a different page.

BRK*Force Break*

Operation: Forces a software interrupt.

In emulation mode, BRK adds 2 to the program counter and pushes it onto the stack; sets the break (B) flag in the status register and pushes it; and loads the program counter with the address in locations \$00FFFE and \$00FFFF.

In native mode, BRK pushes the program bank register onto the stack; adds 2 to the program counter and pushes it onto the stack; pushes the status register; clears the program bank register; and loads the program counter with the address in locations \$00FFE6 and \$00FFE7.

<i>Emulation mode</i>								<i>Native mode</i>							
N	V	I	B	D	I	Z	C	N	V	M	X	D	I	Z	C
.	.	.	1	0	1	0	1	.	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Stack	BRK or BRK nn	00	2 ¹	8 ²

¹BRK is a 1-byte instruction, but the program counter is incremented by 2, thereby allowing for a 1-byte identifier.

²Subtract 1 cycle in emulation mode (E = 1).

BRL*Branch Always Long*Operation: Branches unconditionally to any location in a 64K memory bank. See also **BRA**.

N	V	M	X	D	I	Z	C
.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Program counter relative long	BRL Dis16	82	3	3

BVC*Branch on Overflow Clear*

Operation: Branches on V = 0.

N	V	M	X	D	I	Z	C
.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Program counter relative	BVC Dis8	50	2	2 ¹

¹Add 1 cycle if branch is taken. In emulation mode only, add 1 additional cycle if the branch taken is to a different page.

BVS*Branch on Overflow Set*

Operation: Branches on V = 1.

N	V	M	X	D	I	Z	C
.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Program counter relative	BVS Dis8	70	2	2 ¹

¹Add 1 cycle if branch is taken. In emulation mode only, add 1 additional cycle if the branch taken is to a different page.

CLC*Clear Carry Flag*

Operation: C = 0.

N	V	M	X	D	I	Z	C
.	0

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Implied	CLC	18	1	2

CLD*Clear Decimal Mode*

Operation: Sets D = 0, to select the binary mode.

N	V	M	X	D	I	Z	C
.	.	.	.	0	.	.	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Implied	CLD	D8	1	2

¹Add 1 cycle if branch is taken. In emulation mode only, add 1 additional cycle if the branch taken is to a different page.

CLI*Clear Interrupt Disable Bit*

Operation: Sets I = 0, to enable interrupts.

N	V	M	X	D	I	Z	C
.	0	.	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Implied	CLI	58	1	2

CLV*Clear Overflow Flag*Operation: $V = 0$.

N	V	M	X	D	I	Z	C
.	0

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Implied	CLV	B8	1	2

CMA — See CMP**CMP (or CMA)***Compare Memory and Accumulator*

Operation: Compares by subtracting the operand from the accumulator, but reports the result only in the flags. The accumulator is unaffected.

N	V	M	X	D	I	Z	C
*	*	*

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Immediate	CMP #Num	C9	3 ²	3 ³
Absolute	CMP Loc	CD	3	5 ³
Absolute long	CMP LongLoc	CF	4	6 ³
Absolute indexed with X	CMP Loc,X	DD	3	5 ^{1,3}
Absolute long indexed with X	CMP LongLoc,X	DF	4	6 ³
Absolute indexed with Y	CMP Loc,Y	D9	3	5 ^{1,3}
Direct	CMP DLoc	C5	2	4 ^{3,4}
Direct indexed with X	CMP DLoc,X	D5	2	5 ^{3,4}
Direct indirect	CMP (DLoc)	D2	2	6 ^{3,4}
Direct indirect long	CMP [DLong]	C7	2	7 ^{3,4}
Direct indirect indexed	CMP (DLoc),Y	D1	2	6 ^{1,3,4}
Direct indirect indexed long	CMP [DLong],Y	D7	2	7 ^{3,4}
Direct indexed indirect	CMP (DLoc,X)	C1	2	7 ^{3,4}
Stack relative	CMP Dis8,S	C3	2	5 ³
Stack relative indirect indexed	CMP (Dis8,S),Y	D3	2	8 ³

¹Add 1 cycle if adding index crosses a page boundary.²Subtract 1 byte if M = 1 (8-bit memory/accumulator).³Subtract 1 cycle if M = 1 (8-bit memory/accumulator).⁴Add 1 cycle if direct register low (DL) is not equal to 0.

COP*Coprocessor*

Operation: Forces an interrupt to the vector at locations 00FFE4 and 00FFE5 (native mode) or 00FFF4 and 00FFF5 (emulation mode).

N	V	M	X	D	I	Z	C
.	.	.	.	0	1	.	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Stack	COP nn	02	2	8 ¹

¹Subtract 1 cycle in emulation mode.

CPX*Compare Memory and X Register*

Operation: Compares by subtracting the operand from the X register, but reports the result only in the flags. X is unaffected.

N	V	M	X	D	I	Z	C
*	*	*

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Immediate	CPX #Num	E0	3 ¹	3 ²
Absolute	CPX Loc	ECX	3	5 ²
Direct	CPX DLoc	E4	2	4 ^{2,3}

¹Subtract 1 byte if X = 1 (8-bit index registers).

²Subtract 1 cycle if X = 1 (8-bit index registers).

³Add 1 cycle if direct register low (DL) is not equal to 0.

CPY*Compare Memory and Y Register*

Operation: Compares by subtracting the operand from the Y register, but reports the result only in the flags. Y is unaffected.

N	V	M	X	D	I	Z	C
*	*	*

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Immediate	CPY #Num	C0	3 ¹	3 ²
Absolute	CPY Loc	CC	3	5 ²
Direct	CPY DLoc	C4	2	4 ^{2,3}

¹Subtract 1 byte if X = 1 (8-bit index registers).

²Subtract 1 cycle if X = 1 (8-bit index registers).

³Add 1 cycle if direct register low (DL) is not equal to 0.

DEC*Decrement Memory or Accumulator*

Operation: Subtracts 1 from the operand.

N	V	M	X	D	I	Z	C
*	*	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Accumulator	DEC A or DEA	3A	1	2
Absolute	DEC Loc	CE	3	8 ²
Absolute indexed with X	DEC Loc,X	DE	3	9 ^{1,2}
Direct	DEC DLoc	C6	2	7 ^{2,3}
Direct indexed with X	DEC DLoc,X	D6	2	8 ^{2,3}

¹Add 1 cycle if adding index crosses a page boundary.²Subtract 2 cycles if M = 1 (8-bit memory/accumulator).³Add 1 cycle if direct register low (DL) is not equal to 0.**DEX***Decrement X Register*

Operation: Subtracts 1 from the X register.

N	V	M	X	D	I	Z	C
*	*	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Implied	DEX	CA	1	2

DEY*Decrement Y Register*

Operation: Subtracts 1 from the Y register.

N	V	M	X	D	I	Z	C
*	*	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Implied	DEY	88	1	2

EOR*Exclusive-OR Memory with Accumulator*

Operation: Exclusive-ORs the operand with the accumulator.

N	V	M	X	D	I	Z	C
*	*	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Immediate	EOR #Num	49	3 ²	3 ³
Absolute	EOR Loc	4D	3	5 ³
Absolute long	EOR LongLoc	4F	4	6 ³
Absolute indexed with X	EOR Loc,X	5D	3	5 ^{1,3}
Absolute long indexed with X	EOR LongLoc,X	5F	4	6 ³
Absolute indexed with Y	EOR Loc,Y	59	3	5 ^{1,3}
Direct	EOR DLoc	45	2	4 ^{3,4}
Direct indexed with X	EOR DLoc,X	55	2	5 ^{3,4}
Direct indirect	EOR (DLoc)	52	2	6 ^{3,4}
Direct indirect long	EOR [DLong]	47	2	7 ^{3,4}
Direct indirect indexed	EOR (DLoc),Y	51	2	6 ^{1,3,4}
Direct indirect indexed long	EOR [DLong],Y	57	2	7 ^{3,4}
Direct indexed indirect	EOR (DLoc,X)	41	2	7 ^{3,4}
Stack relative	EOR Dis8,S	43	2	5 ³
Stack relative indirect indexed	EOR (Dis8,S),Y	53	2	8 ³

¹Add 1 cycle if adding index crosses a page boundary.²Subtract 1 byte if M = 1 (8-bit memory/accumulator).³Subtract 1 cycle if M = 1 (8-bit memory/accumulator).⁴Add 1 cycle if direct register low (DL) is not equal to 0.**INC***Increment Memory or Accumulator*

Operation: Adds 1 to the operand.

N	V	M	X	D	I	Z	C
*	*	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Accumulator	INC A or INA	1A	1	2
Absolute	INC Loc	EE	3	8 ²
Absolute indexed with X	INC Loc,X	FE	3	9 ^{1,2}
Direct	INC DLoc	E6	2	7 ^{2,3}
Direct indexed with X	INC DLoc,X	F6	2	8 ^{2,3}

¹Add 1 cycle if adding index crosses a page boundary.²Subtract 2 cycles if M = 1 (8-bit memory/accumulator).³Add 1 cycle if direct register low (DL) is not equal to 0.

INX*Increment X Register*

Operation: Adds 1 to the X register.

N	V	M	X	D	I	Z	C
*	*	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Implied	INX	E8	1	2

INY*Increment Y Register*

Operation: Adds 1 to the Y register.

N	V	M	X	D	I	Z	C
*	*	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Implied	INY	C8	1	2

JML*Jump Long*Operation: Transfers to a new location indirectly, using an address it obtains from a 3-byte pointer in memory. See also **JMP**.

N	V	M	X	D	I	Z	C
.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Absolute indirect	JML (Loc)	DC	3	6

JMP*Jump*

Operation: Transfers to the specified target location. Note that JMP's absolute indirect mode obtains a 2-byte address (PC contents) from the indirect location. Compare this with JML, which obtains a 3-byte address (PC plus PBR) from the location.

N V M X D I Z C

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Absolute	JMP Loc	4C	3	3
Absolute long	JMP LongLoc	5C	4	4
Absolute indirect	JMP (Loc)	6C	3	5
Absolute indexed indirect	JMP (Loc,X)	7C	3	6

JSL*Jump to Subroutine Long*

Operation: Saves the PC and program bank register (PBR) on the stack, then transfers to the specified target location. The target can be anywhere in memory. Compare JSL with JSR, which can only transfer within the current bank. An RTL instruction is used to return from the subroutine.

N V M X D I Z C

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Absolute long	JSL LongLoc	22	4	8

JSR*Jump to Subroutine*

Operation: Saves the PC on the stack, then transfers to the specified target location. The target must be within the current program bank. Compare JSR with JSL, which can transfer anywhere in memory. An RTS instruction is used to return from the subroutine.

N V M X D I Z C

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Absolute	JSR Loc	20	3	6
Absolute indexed indirect	JSR (DLoc,X)	FC	3	6

LDA*Load Accumulator*

Operation: Loads the accumulator with the contents of the operand.

N	V	M	X	D	I	Z	C
*	*	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Immediate	LDA #Num	A9	3 ²	3 ³
Absolute	LDA Loc	AD	3	5 ³
Absolute long	LDA LongLoc	AF	4	6 ³
Absolute indexed with X	LDA Loc,X	BD	3	5 ^{1,3}
Absolute long indexed with X	LDA LongLoc,X	BF	4	6 ³
Absolute indexed with Y	LDA Loc,Y	B9	3	5 ^{1,3}
Direct	LDA DLoc	A5	2	4 ^{3,4}
Direct indexed with X	LDA DLoc,X	B5	2	5 ^{3,4}
Direct indirect	LDA (DLoc)	B2	2	6 ^{3,4}
Direct indirect indexed	LDA (DLoc),Y	B1	2	6 ^{1,3,4}
Direct indirect indexed long	LDA [DLong],Y	B7	2	7 ^{3,4}
Direct indexed indirect	LDA (DLoc,X)	A1	2	7 ^{3,4}
Stack relative	LDA Dis8,S	A3	2	5 ³
Stack relative indirect indexed	LDA (Dis8,S),Y	B3	2	8 ³

¹Add 1 cycle if adding index crosses a page boundary.²Subtract 1 byte if M = 1 (8-bit memory/accumulator).³Subtract 1 cycle if M = 1 (8-bit memory/accumulator).⁴Add 1 cycle if direct register low (DL) is not equal to 0.**LDX***Load X register*

Operation: Loads the X register with the contents of the operand.

N	V	M	X	D	I	Z	C
*	*	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Immediate	LDX #Num	A2	2 ²	3 ³
Absolute	LDX Loc	AE	3	5 ³
Absolute indexed with Y	LDX Loc,Y	BE	3	5 ^{1,3}
Direct	LDX DLoc	A6	2	4 ^{3,4}
Direct indexed with Y	LDX DLoc,Y	B6	2	5 ^{3,4}

¹Add 1 cycle if adding index crosses a page boundary.²Subtract 1 byte if X = 1 (8-bit index registers).³Subtract 1 cycle if X = 1 (8-bit index registers).⁴Add 1 cycle if direct register low (DL) is not equal to 0.

LDY*Load Y register*

Operation: Loads the Y register with the contents of the operand.

N	V	M	X	D	I	Z	C
*	*	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Immediate	LDY #Num	A0	3 ²	3 ³
Absolute	LDY Loc	AC	3	5 ³
Absolute indexed with X	LDY Loc,X	BC	3	5 ^{1,3}
Direct	LDY DLoc	A4	2	4 ^{3,4}
Direct indexed with X	LDY DLoc,X	B4	2	5 ^{3,4}

¹Add 1 cycle if adding index crosses a page boundary.²Subtract 1 byte if X = 1 (8-bit index registers).³Subtract 1 cycle if X = 1 (8-bit index registers).⁴Add 1 cycle if direct register low (DL) is not equal to 0.**LSR***Logical Shift Right*

Operation: Shifts the operand right one bit position. It puts the displaced bit in the carry flag (C) and puts a 0 in the vacated bit position. See also ASL, which shifts operands left.

N	V	M	X	D	I	Z	C
0	*	*

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Accumulator	LSR A	4A	1	2
Absolute	LSR Loc	4E	3	8 ²
Absolute indexed with X	LSR Loc,X	5E	3	9 ^{1,2}
Direct	LSR DLoc	46	2	7 ^{2,3}
Direct indexed with X	LSR DLoc,X	56	2	8 ^{2,3}

¹Add 1 cycle if adding index crosses a page boundary.²Subtract 2 cycles if M = 1 (8-bit memory/accumulator).³Add 1 cycle if direct register low (DL) is not equal to 0.

MVN*Move Block Negative*

Operation: Moves a block of bytes to a lower-numbered address. Before executing MVN, set up the following registers:

- X = Address of beginning of source block
- Y = Address of beginning of destination block
- A = Number of bytes to be copied, less 1

N V M X D I Z C

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Block move	MVN ss,DLoc	54	3	7*n

Here, *ss* and *DLoc* are the hexadecimal numbers of the source and destination banks, while *n* is the number of bytes moved.

MVP*Move Block Positive*

Operation: Moves a block of bytes to a higher-numbered address. Before executing MVP, set up the following registers:

- X = Address of end of source block
- Y = Address of end of destination block
- A = Number of bytes to be copied, less 1

N V M X D I Z C

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Block move	MVP ss,DLoc	44	3	7*n

Here, *ss* and *DLoc* are the hexadecimal numbers of the source and destination banks, while *n* is the number of bytes moved.

NOP*No Operation*

Operation: Does nothing except advance the program counter.

N V M X D I Z C

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Implied	NOP	EA	1	2

ORA*OR Memory with Accumulator*

Operation: ORs the operand with the accumulator.

N	V	M	X	D	I	Z	C
*	*	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Immediate	ORA #Num	09	3 ²	3 ³
Absolute	ORA Loc	0D	3	5 ³
Absolute long	ORA LongLoc	0F	4	6 ³
Absolute indexed with X	ORA Loc,X	1D	3	5 ^{1,3}
Absolute long indexed with X	ORA LongLoc,X	1F	4	6 ³
Absolute indexed with Y	ORA Loc,Y	19	3	5 ^{1,3}
Direct	ORA DLoc	05	2	4 ^{3,4}
Direct indexed with X	ORA DLoc,X	15	2	5 ^{3,4}
Direct indirect	ORA (DLoc)	12	2	6 ^{3,4}
Direct indirect long	ORA [DLoc]	07	2	7 ³
Direct indirect indexed	ORA (DLoc),Y	11	2	6 ^{1,3,4}
Direct indirect indexed long	ORA [DLoc],Y	17	2	7 ^{3,4}
Direct indexed indirect	ORA (DLoc,X)	01	2	7 ^{3,4}
Stack relative	ORA Dis8,S	03	2	5 ³
Stack relative indirect indexed	ORA (Dis8,S),Y	13	2	8 ³

¹Add 1 cycle if adding index crosses a page boundary.²Subtract 1 byte if M = 1 (8-bit memory/accumulator).³Subtract 1 cycle if M = 1 (8-bit memory/accumulator).⁴Add 1 cycle if direct register low (DL) is not equal to 0.**PEA***Push Effective Absolute Address onto Stack
or Push Immediate Word onto Stack*

Operation: Pushes a 16-bit operand onto the stack.

N	V	M	X	D	I	Z	C
.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Stack	PEA Loc	F4	3	5

PEI

*Push Effective Indirect Address onto Stack
or Push Direct Page Word onto Stack*

Operation: Adds the offset to the direct page register, then pushes the 2 bytes at that address onto the stack.

N V M X D I Z C
· · · · · · · ·

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Stack	PEI (DLoc)	D4	2	6 ¹

¹Add 1 cycle if data register low (DL) is not equal to 0.

PER

Push Effective Program Counter Relative Address onto Stack

Operation: Adds a 16-bit constant to the program counter, then pushes the resulting value onto the stack.

N V M X D I Z C
· · · · · · · ·

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Stack	PER #Num	62	3	6

PHA

Push Accumulator onto Stack

Operation: Pushes A onto the stack.

N V M X D I Z C
· · · · · · · ·

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Stack	PHA	48	1	4 ¹

PHB

Push Data Bank Register onto Stack

Operation: Pushes the DBR onto the stack.

N V M X D I Z C
· · · · · · · ·

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Stack	PHB	8B	1	3

420 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

PHD

Push Direct Register onto Stack

Operation: Pushes the D register onto the stack.

N V M X D I Z C
· · · · · · · ·

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Stack	PHD	0B	1	3

PHK

Push Program Bank Register onto Stack

Operation: Pushes the PBR onto the stack.

N V M X D I Z C
· · · · · · · ·

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Stack	PHK	4B	1	3

PHP

Push Processor Status Register onto Stack

Operation: Pushes the P register onto the stack.

N V M X D I Z C
· · · · · · · ·

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Stack	PHP	08	1	3

PHX

Push X Register onto Stack

Operation: Pushes X onto the stack.

N V M X D I Z C
· · · · · · · ·

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Stack	PHX	DA	1	4 ¹

¹Subtract 1 cycle if X = 1 (8-bit index registers).

PHY*Push Y Register onto Stack*

Operation: Pushes Y onto the stack.

N	V	M	X	D	I	Z	C
.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Stack	PHY	5A	1	4 ¹

¹Subtract 1 cycle if X = 1 (8-bit index registers).**PLA***Pull Accumulator from Stack*

Operation: Retrieves A from the stack.

N	V	M	X	D	I	Z	C
*	*	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Stack	PLA	68	1	5 ¹

PLB*Pull Data Bank Register from Stack*

Operation: Retrieves the DBR from the stack.

N	V	M	X	D	I	Z	C
*	*	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Stack	PHB	AB	1	4

PLD*Pull Direct Register from Stack*

Operation: Retrieves D from the stack.

N	V	M	X	D	I	Z	C
*	*	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Stack	PLD	2B	1	4

PLP*Pull Processor Status Register from Stack*

Operation: Retrieves P from the stack.

N	V	M	X	D	I	Z	C
*	*	*	*	*	*	*	*

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Stack	PLP	28	1	4

PLX*Pull X Register from Stack*

Operation: Retrieves X from the stack.

N	V	M	X	D	I	Z	C
*	*	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Stack	PLX	FA	1	5 ¹

PLY*Pull Y Register from Stack*

Operation: Retrieves Y from the stack.

N	V	M	X	D	I	Z	C
*	*	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Stack	PLY	7A	1	5 ¹

REP*Reset Processor Status Bits*

Operation: ANDs the processor status register with the complement of an immediate byte.
Thus, each 1 bit in the operand clears the corresponding bit in P.

	N	V	M	X	D	I	Z	C
(Native mode)	*	*	*	*	*	*	*	*
(Emulation mode)	*	*	.	.	*	*	*	*

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Immediate	REP #Num	C2	2	3

ROL*Rotate Left*

Operation: Rotates the operand left one bit position. It puts the contents of the carry flag (C) in the vacated bit position, then puts the displaced bit in C. See also **ROR**, which rotates operands right.

N	V	M	X	D	I	Z	C
*	*	*

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Accumulator	ROL A	2A	1	2
Absolute	ROL Loc	2E	3	8 ²
Absolute indexed with X	ROL Loc,X	3E	3	9 ^{1,2}
Direct	ROL DLoc	26	2	7 ^{2,3}
Direct indexed with X	ROL DLoc,X	36	2	8 ^{2,3}

¹Add 1 cycle if adding index crosses a page boundary.

²Subtract 2 cycles if M = 1 (8-bit memory/accumulator).

³Add 1 cycle if direct register low (DL) is not equal to 0.

ROR*Rotate Right*

Operation: Rotates the operand right one bit position. It puts the contents of the carry flag (C) in the vacated bit position, then puts the displaced bit in C. See also **ROL**, which rotates operands left.

N	V	M	X	D	I	Z	C
*	*	*

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Accumulator	ROR A	6A	1	2
Absolute	ROR Loc	6E	3	8 ²
Absolute indexed with X	ROR Loc,X	7E	3	9 ^{1,2}
Direct	ROR DLoc	66	2	7 ²
Direct indexed with X	ROR DLoc,X	76	2	8 ^{2,3}

¹Add 1 cycle if adding index crosses a page boundary.

²Subtract 2 cycles if M = 1 (8-bit memory/accumulator).

³Add 1 cycle if direct register low (DL) is not equal to 0.

RTI*Return from Interrupt*

Operation: Retrieves the contents of the processor status register (P), program counter (PC), and program bank register (PBR) from the stack.

N V M X D I Z C
* * * * *

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Stack	RTI	40	1	7

RTL*Return from Subroutine Long*

Operation: Retrieves the contents of the program counter (PC) and program bank register (PBR) from the stack. That is, it undoes the work of a JSL instruction.

N V M X D I Z C
.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Stack	RTL	6B	1	6

RTS*Return from Subroutine*

Operation: Retrieves the contents of the program counter (PC) from the stack. That is, it undoes the work of a JSR instruction.

N V M X D I Z C
.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Stack	RTS	60	1	6

SBC*Subtract from Accumulator with Borrow*

Operation: Subtract the operand and the complement of the carry flag from the accumulator.

In effect, the operation is $A = A - M + C$.

N	V	M	X	D	I	Z	C
*	*	*	*

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Immediate	SBC #Num	E9	3 ²	3 ³
Absolute	SBC Loc	ED	3	5 ³
Absolute long	SBC LongLoc	EF	4	6 ³
Absolute indexed with X	SBC Loc,X	FD	3	5 ^{1,3}
Absolute long indexed with X	SBC LongLoc,X	FF	4	6 ³
Absolute indexed with Y	SBC Loc,Y	F9	3	5 ^{1,3}
Direct	SBC DLoc	E5	2	4 ^{3,4}
Direct indexed with X	SBC DLoc,X	F5	2	5 ^{3,4}
Direct indirect	SBC (DLoc)	F2	2	6 ^{3,4}
Direct indirect long	SBC [DLong]	E7	2	7 ^{3,4}
Direct indirect indexed	SBC (DLoc),Y	F1	2	6 ^{1,3,4}
Direct indirect indexed long	SBC [DLong],Y	F7	2	7 ^{3,4}
Direct indexed indirect	SBC (DLoc,X)	E1	2	7 ^{3,4}
Stack relative	SBC Dis8,S	E3	2	5 ³
Stack relative indirect indexed	SBC (Dis8,S),Y	F3	2	8 ³

¹Add 1 cycle if adding index crosses a page boundary.²Subtract 1 byte if M = 1 (8-bit memory/accumulator).³Subtract 1 cycle if M = 1 (8-bit memory/accumulator).⁴Add 1 cycle if direct register low (DL) is not equal to 0.**SEC***Set Carry Flag*

Operation: C = 1.

N	V	M	X	D	I	Z	C
.	1

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Implied	SEC	38	1	2

SED*Set Decimal Mode*

Operation: Sets D = 1, to select the decimal mode.

N	V	M	X	D	I	Z	C
.	.	.	.	1	.	.	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Implied	SED	F8	1	2

SEI*Set Interrupt Disable Bit*

Operation: Sets I = 1, to disable or lock out interrupts.

N	V	M	X	D	I	Z	C
.	1	.	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Implied	SEI	78	1	2

SEP*Set Processor Status Bits*

Operation: ORs the processor status register with an immediate byte. Thus, each 1 bit in the operand sets the corresponding bit in P.

	N	V	M	X	D	I	Z	C
(Native mode)	*	*	*	*	*	*	*	*
(Emulation mode)	*	*	.	.	*	*	*	*

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Immediate	SEP #nn	E2	2	3

STA*Store Accumulator*

Operation: Stores the contents of the accumulator at the specified memory location.

N	V	M	X	D	I	Z	C
.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Absolute	STA Loc	8D	3	5 ²
Absolute long	STA LongLoc	8F	4	6 ²
Absolute indexed with X	STA Loc,X	9D	3	6 ^{1,2}
Absolute long indexed with X	STA LongLoc,X	9F	4	6 ²
Absolute indexed with Y	STA Loc,Y	99	3	6 ^{1,2}
Direct	STA DLoc	85	2	4 ^{2,3}
Direct indexed with X	STA DLoc,X	85	2	5 ^{2,3}
Direct indirect	STA (DLoc)	92	2	6 ^{2,3}
Direct indirect long	STA [DLong]	87	2	7 ^{2,3}
Direct indirect indexed	STA (DLoc),Y	91	2	7 ^{1,2,3}
Direct indirect indexed long	STA [DLong],Y	97	2	7 ^{2,3}
Direct indexed indirect	STA (DLoc,X)	81	2	7 ^{2,3}
Stack relative	STA Dis8,S	83	2	5 ²
Stack relative indirect indexed	STA (Dis8,S),Y	93	2	8 ²

¹Add 1 cycle if adding index crosses a page boundary.²Subtract 1 cycle if M = 1 (8-bit memory/accumulator).³Add 1 cycle if direct register low (DL) is not equal to 0.**STP***Stop the Clock*

Operation: Stops the processor until it receives a hardware reset.

N	V	M	X	D	I	Z	C
.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Implied	STP	DB	1	3

STX*Store X Register*

Operation: Stores the contents of the X register at the specified memory location.

N	V	M	X	D	I	Z	C
.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Absolute	STX Loc	8E	3	5 ¹
Direct	STX DLoc	86	2	4 ^{1,2}
Direct indexed with Y	STX DLoc,Y	96	2	5 ¹

¹Subtract 1 cycle if X = 1 (8-bit index registers).²Add 1 cycle if direct register low (DL) is not equal to 0.**STY***Store Y Register*

Operation: Stores the contents of the Y register at the specified memory location.

N	V	M	X	D	I	Z	C
.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Absolute	STY Loc	8C	3	5 ¹
Direct	STY DLoc	84	2	4 ^{1,2}
Direct indexed with X	STY DLoc,X	94	2	5 ^{1,2}

¹Subtract 1 cycle if X + 1 (8-bit index registers).²Add 1 cycle if direct register low (DL) is not equal to 0.**STZ***Store Zero*

Operation: Stores 0 at the specified memory location.

N	V	M	X	D	I	Z	C
.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Absolute	STZ Loc	9C	3	5 ¹
Absolute indexed with X	STZ Loc,X	9E	3	6 ¹
Direct	STZ DLoc	64	2	4 ^{1,2}
Direct indexed with X	STZ DLoc,X	74	2	5 ^{1,2}

¹Subtract 1 cycle if M = 1 (8-bit memory/accumulator).²Add 1 cycle if direct register low (DL) is not equal to 0.³Add 1 cycle if adding index crosses a page boundary.

SWA — See XBA

TAD — See TCD

TAS — See TCS

TAX*Transfer Accumulator to X Register*

Operation: Copies the contents of A into X.

N	V	M	X	D	I	Z	C
*	*	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Implied	TAX	AA	1	2

TAY*Transfer Accumulator to Y Register*

Operation: Copies the contents of A into Y.

N	V	M	X	D	I	Z	C
*	*	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Implied	TAY	A8	1	2

TCD (or TAD)*Transfer C Accumulator to Direct Register*

Operation: Copies the contents of C (the 16-bit accumulator) into D.

N	V	M	X	D	I	Z	C
*	*	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Implied	TCD	5B	1	2

430 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

TCS (or TAS)

Transfer C Accumulator to Stack Pointer

Operation: Copies the contents of C (the 16-bit accumulator) into S.

N V M X D I Z C

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Implied	TCS	1B	1	2

TDA — See TDC

TDC (or TDA)

Transfer Directo Register to C Accumulator

Operation: Copies the contents of D into C, the 16-bit accumulator.

N V M X D I Z C
 * *

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Implied	TDC	7B	1	2

TRB

Test and Reset Bits

Operation: ANDs the memory operand with the complement of the accumulator. Each 1 bit in the accumulator becomes 0 in the operand; each 0 bit in the accumulator leaves the corresponding operand bit unchanged.

N V M X D I Z C
 *

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Absolute	TRB Loc	1C	3	5 ¹
Direct	TRB DLoc	14	2	4 ^{1,2}

¹Subtract 1 cycle if M = 1 (8-bit memory/accumulator).

²Add 1 cycle if direct register low (DL) is not equal to 0.

TSA — See TSC

TSB*Test and Reset Bits*

Operation: ORs the memory operand with the accumulator. Each 1 bit in the accumulator becomes 1 in the operand; each 0 bit in the accumulator leaves the corresponding operand bit unchanged.

N	V	M	X	D	I	Z	C
.	*	.

Addressing Mode	Assembler Format		Opcode	Bytes	Cycles
Absolute	TSB	Loc	0C	3	5 ¹
Direct	TSB	DLoc	04	2	4 ^{1,2}

¹Subtract 1 cycle with M = 1 (8-bit memory/accumulator).

²Add 1 cycle if direct register low (DL) is not equal to 0.

TSC (or TSA)*Transfer Stack Pointer to C Accumulator*

Operation: Copies the contents of S into C, the 16-bit accumulator.

N	V	M	X	D	I	Z	C
*	*	.

Addressing Mode	Assembler Format		Opcode	Bytes	Cycles
Implied	TSC		3B	1	2

TSX*Transfer Stack Pointer to X Register*

Operation: Copies the contents of S into X.

N	V	M	X	D	I	Z	C
*	*	.

Addressing Mode	Assembler Format		Opcode	Bytes	Cycles
Implied	TSX		BA	1	2

432 APPLE IIGS ASSEMBLY LANGUAGE PROGRAMMING

TXA

Transfer X Register to Accumulator

Operation: Copies the contents of X into A.

N	V	M	X	D	I	Z	C
*	*	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Implied	TXA	8A	1	2

TXS

Transfer X Register to Stack Pointer

Operation: Copies the contents of X into S.

N	V	M	X	D	I	Z	C
.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Implied	TXS	9A	1	2

TXY

Transfer X Register to Y Register

Operation: Copies the contents of X into Y.

N	V	M	X	D	I	Z	C
*	*	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Implied	TXY	9B	1	2

TYA

Transfer Y Register to Accumulator

Operation: Copies the contents of Y into A.

N	V	M	X	D	I	Z	C
*	*	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Implied	TYA	98	1	2

TYX*Transfer Y Register to X Register*

Operation: Copies the contents of Y into X.

N	V	M	X	D	I	Z	C
*	*	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Implied	TYX	BB	1	2

WAI*Wait for Interrupt*

Operation: Puts the processor into a low-power idle state until it receives an external hardware interrupt.

N	V	M	X	D	I	Z	C
.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Implied	WAI	CB	1	3

XBA*Exchange B and A Bytes of Accumulator***SWA***Swap Accumulator Bytes*

Operation: Swaps B (high byte) and A (low byte).

N	V	M	X	D	I	Z	C
*	*	.

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Implied	XBA	EB	1	3

XCE*Exchange Carry and Emulation Bits*

Operation: Swaps the processor status register's C and E bits. If C is 1, the processor enters the 8-bit emulation mode; if C is 0, it enters the full 16-bit native mode.

N	V	M	X	D	I	Z	C
.	E

Addressing Mode	Assembler Format	Opcode	Bytes	Cycles
Implied	XCE	FB	1	2

APPENDIX D

Requirements for Using Tool Sets

With the exception of the Tool Locator, each tool set requires other tool sets to be active. Table D-1 shows which tool sets you must start to use a particular tool set. The tool sets are listed in the order in which you must start them.

Shut down the tool sets in the following order:

- Desk Manager
- Font Manager, Print Manager, and other tool sets not listed here
- Menu Manager
- Window Manager
- Control Manager
- Dialog Manager
- Event Manager
- LineEdit
- QuickDraw
- Miscellaneous Tools
- Memory Manager (after freeing all allocated memory)
- Tool Locator

Loading RAM-Based Tools from Disk

Before starting the Window Manager, you must call LoadTools to read any needed RAM-based tools from disk. This includes not only the tool sets that are entirely on disk (e.g., the Window, Menu, and Control Managers), but also additional tools for some of the tool sets in ROM, such as QuickDraw's oval-drawing routines. If your program needs these tools, it must load them into memory. LoadTools requires a list of the identification numbers and minimum version numbers for the tool sets you want it to load from disk. Table D-2 lists the tool set numbers.

Table D-1

To use ↓	You must start the following tool sets:										
	Tool Locator	Memory Manager	Misc. Tools	QuickDraw	Event Manager	Window Manager	Control Manager	LineEdit	Dialog Manager	Menu Manager	Desk Manager
Tool Locator	•										
Memory Manager	•	•									
Misc. Tools	•	•	•								
QuickDraw	•	•	•	•							
Event Manager	•	•	•	•	•						
Window Manager*	•	•	•	•	•	•	•			•	
Control Manager	•	•	•	•	•	•	•				
LineEdit Dialog Manager	•	•	•	•	•	•	•	•	•	•	
Menu Manager	•	•	•	•	•	•	•			•	
Desk Manager	•	•	•	•		•	•	•	•	•	•

*The Control Manager and Menu Manager are only required if you want to use the Window Manager's TaskMaster.

Table D-2

ROM-Based Tools Sets	
1	Tool Locator
2	Memory Manager
3	Miscellaneous Tools
4	QuickDraw II
5	Desk Manager
6	Event Manager
7	Scheduler
8	Sound Manager
9	Apple Desktop Bus
10	SANE
11	Integer Math Tools
12	Text Tools
13	(Used internally)

RAM-Based Tools Sets	
14	Window Manager
15	Menu Manager
16	Control Manager
17	Loader
18	Quickdraw Auxiliary Routines
19	Print Manager
20	LineEdit
21	Dialog Manager
22	Scrap Manager
23	Standard File Operations
24	Disk Utilities
25	Note Synthesizer
26	Note Sequencer
27	Font Manager
28	List Manager

Working Space for Tool Sets

Many tool sets require working space in bank 0, and you must make the Memory Manager allocate it with a `NewHandle` call. Specifically, QuickDraw requires three pages (or \$300 bytes), while the following sets each require one page (\$100 bytes):

Event Manager (shares its page with the Window Manager)

Control Manager (shares its page with the Dialog Manager)

Menu Manager

LineEdit

Font Manager

Print Manager

Sound Manager

Standard File Operations

SANE

Index

; (comment specifier), 35
\$ (hexadecimal specifier), 34-35
= (ProDOS wildcard character), 154
% (binary specifier), 34-35
(APW system prompt), 47
&LAB (symbolic label in macro definition), 134
___ (underscore prefix for tool calls), 166

65816 instruction set, 90-131,
 401-433
 list of, 91-93
 See also *Quick Index*
65816 internal registers, 13-19
65816 operand addressing modes,
 summary of, 74

A

A register (Accumulator), 15
Absolute addressing modes, 75-76
Absolute indexed addressing modes,
 78
Absolute indexed indirect addressing
 mode, 79-80

Absolute indirect addressing mode,
 77-78
Absolute long indexed with X
 addressing mode, 79
Accumulator (A) register, 15
Accumulator addressing mode, 75
Activate (window) event, 274
Active controls, 359
Active window, 299
Add and subtract macros, 146,
 150
Adding binary numbers, 4
Addition, 100-101
Address bus, 10
Addressing modes, operand formats
 for, 86
Addressing modes, summary tables
 of, 74, 86-87
AINCLUDE subdirectory on APW
 disk, 143
Alert
 box, 362, 364-367
 stages of an, 366
 template, 395-396
 window, 294-295
Alternate mnemonics for 65816
 instructions, 91, 93

Animated pixel image program
 (ANIMATE), 254-255
 Animation, 249, 253-254
 Apple II graphics modes, 191-192
 Apple IIGS
 internal architecture of, 19-22
 memory in, 22-25
 Monitor, 25-26
 Programmer's Workshop macros,
 143-154
 Programmer's Workshop,
 starting, 47
 Programmer's Workshop
 commands, 54-58
 super high-resolution graphics
 modes, 192
 Toolbox, 26, 155-190
 Apple menu, 327, 331, 336
 Application program, general
 structure of a, 168
 APW — See *Apple IIGS
 Programmer's Workshop*
 Arcs, drawing, 223
 Arithmetic instructions, 99-105
 Arithmetic operators, 43, 45
 Arithmetic, signed, 104
 Arranging two numbers in increasing
 order, 110
 ASCII character table, 400
 ASM65816 command, 54
 ASML (assemble and link)
 command, 55
 ASMLG (assemble, link, and go)
 command, 55
 ASSEMBLE command, 55
 Assembler, 27
 Assembler directives, 35-43
 See also *Quick Index*

Assembler directives, summary
 tables of, 37, 43
 Assembly language program, steps
 in developing an, 28-31
 Attributes, pen, 198
 Auto-key event, 274
 Automating the assembly process,
 62

B

B (Break command) flag, 19
 Banks \$E0 and \$E1 in Apple IIGS,
 23-24
 Banks, memory, 11
 BEEP (speaker-beeping program),
 48-49
 Binary (base 2) numbering, 1-7
 Binary digits (bits), 2
 Binary numbers, adding, 4
 Binary-coded decimal (BCD)
 numbers, 99-101, 103
 Bit (binary digit), 2
 Bit manipulation instructions,
 119-123
 Bit maps, Macintosh, 238-240
 Bit-testing instructions, 122-123
 Block move addressing mode, 85
 Block move instructions, 96-99
 Blocks in memory, 156
 Borrow in subtraction, 103
 BoundsRect, QuickDraw, 242, 245
 Bowtie-drawing program
 (BOWTIE), 225, 227
 Branching directives, 140-141
 Break command (B) flag, 19
 Breakpoints, 71-72

Breakpoints subdisplay, debugger,
66
BRK (breakpoint flag) on debugger
screen, 65
BUILD files, 62
Button-reading tool call, 286
Buttons, 159, 357
Byte (8-bit value), 4
Byte Works, Inc., The, 27
Bytes, packed, of BCD digits,
100

C

Calculation calls for shapes, 225-226
Calling conventions for tools,
166-168
Carry (C) flag, 17-18
CATALOG command, 54
Caution Alert box, 365
CDAs — See Classic desk
accessories
CHANGE command, 54
Check boxes, 159, 358
Check error macro, 147, 154
Circles, drawing, 224
Classic desk accessories (CDAs),
162
ClipRgn (clip region), Quickdraw,
246
Clock speeds, 12
Close box (go-away region),
295-296, 302, 311
CMOS, 10
Code segment, 36
Codes, event, 277-279

Color tables, 205, 266-269
Color-dithering program
(DITHERS), 232-233
Colors, 202, 205-207
320 mode, 205-206
640 mode, 206-208
Comment field, 35
Common programming errors,
189-190
Compare instructions, 105, 109-111
using branch instructions with,
109-111
Conceptual drawing space,
Quickdraw, 192-194
Conditional transfer (branch)
instructions, 107-111
Content region of a window, 295
Control Manager, 159-160, 356
Control Panel, 162, 275
Control transfer instructions,
106-111
Controls, on-screen, 159-160
Controls, predefined, 357-359
Conversion commands, debugger,
69, 71
Converting decimal values to binary,
3
Converting hexadecimal numbers to
decimal, 399
Copy (pen mode), 198
COPY file command, 56
Copying between programs, 186
Correcting typing errors, 50-53
CREATE subdirectory command,
56
Creating macro libraries, 142-143
Crystal, 12

D

- D (decimal mode) flag, 19
- D (Direct) register, 16
- Data Bank Register (DBR), 15
- Data bus, 10
- Data formats for numeric values, 99-100
- Data transfer instructions, 94-99
- Date and time operations, 254, 259-266
- DEBUG command, 57
- DebugD (direct page on debugger screen), 65
- Debugger, 29-31, 63-72
 - commands, summary table of, 67-69
 - leaving the, 72
 - starting the, 63-64
- Decimal mode (D) flag, 19
- Decimal numbers, 99-100
- Decimal values, converting to binary, 3
- Decimal values, converting to hexadecimal, 399
- Decision sequence, three-way, 110-111
- Decrement instructions, 104-105
- Default color table — See *Standard color table*
- Define macros, 146, 150-151
- DELAY subroutine, 262, 264, 266
- Delays, generating, 261-266
- DELETE file command, 56
- Desk accessories, 161-162
- Desk accessory event, 275
- Desk Manager, 161-162, 275
- Desk scrap, 162
- Dialog box, 161, 362-364, 366-367
- Dialog Manager, 161, 362, 367-372
 - item lists, 367-372
 - item types, 368-372
- Dials, 358
- Digital Oscillator Chip (DOC), 162
- Direct addressing mode, 80-81
- Direct indexed addressing modes, 82
- Direct indexed indirect addressing mode, 83
- Direct indirect addressing modes, 81-82
- Direct indirect indexed addressing modes, 82-83
- Direct page, 11
- Direct register, 16
- Directives, assembler, 35-42
 - summary table of, 37, 43
 - See also *Quick Index*
- Disabled menu, 329
- Disassembly commands, debugger, 69-70
- Disassembly subdisplay, debugger, 66
- Displaying a pixel image, 247-249
- Displaying text, 232, 236-237
- Dithering in 640 mode, 207-209, 232
- DITHERS (color-dithering program), 232-233
- DOC (Digital Oscillator Chip), 162
- Document window, 294-295
- Double high- and low-resolution graphics modes, 192
- Drawing
 - arcs, 223
 - circles, 224

- lines, 210-211
- ovals, 223-224
- polygons, 218-223
- rectangles, 211-218
- regions, 224-225
- Drawing space, Quickdraw
 - conceptual, 192-194

E

- Edit (selection in menu), 327
- EDIT command, 54
- Editing commands, debugger, 67, 69
- Editor, 29, 50-54
 - leaving the, 53-54
 - starting the, 50
- Editor commands, 51-52
- Effective address calculations, 86
- Elapsed time in event record, 279
- Emulation mode, 13, 17, 19,
 - 127-128
 - status bits in, 17, 19
 - switching to, 127-128
- ENABLE command, 57
- Enabled menu, 328
- Ensoniq Digital Oscillator Chip (DOC), 162
- Equate directives, 40-41
- Errors, correcting, 50-53
- Errors, programming, common, 189-190
- Event
 - codes, 277-279
 - definition of, 160, 271
 - loop, 272-273
 - mask, 283-284

- message, 279
- priorities, 276-277
- queue, 160, 271, 276-277
- record, 271, 277-281
- types, 273-276
- Event Manager, 160-161, 271, 281-286
- EXE (shell load) files, 62
- EXEC command, 54
- EXEC files, 62

F

- Fast Processor Interface (FPI) chip, 20
- File (selection in menu), 327
- File control directives, 39
- FILETYPE command, 57, 62
- Firmware, 20
- Fixed blocks in memory, 156
- Flags, status, 17
- Flowchart, 28
- Font Manager, 162, 359
- Format disk (INIT) command, 57
- Frame (rectangle boundary), 211

G

- General-purpose registers, 15
- Generating delays, 261-266
- Global labels, 38
- Go-away region (close box), 310
- GrafPort, Quickdraw, 242-247
 - record, 244
- Graphics modes, 191-192
- Grow box, 296-297, 302

H

Handle (pointer to a pointer), 157, 169-171
 HELP command, 54
 Hertz (cycles per second), 12
 Hexadecimal (base 16) numbering, 1, 7-8
 Hexadecimal to decimal conversion, 399
 High-resolution graphics mode, 191

instructions, 128-131
 request, 13
 requests, disabling, 19
 service routine (handler), 128
 vectors, 13, 128
 software, 130
 IRQ disable (I) flag, 19
 Item
 lists, 367-372
 template, 395-396
 types, 368-372

I

Immediate addressing mode, 74-75
 Implied addressing mode, 75
 Inactive controls, 360
 Inactive window, 299
 Increment and decrement
 instructions, 104-105
 Index register select bit (X), 19
 Index registers (X and Y), 15
 Information bar, 298-299
 INIT (format disk) command, 57
 Instruction set, 65816, 90-131, 401-433
 See also *Quick Index*
 Instruction set, 65816, list of, 91-93
 Integer Math Tools, 163
 Integers, 168
 Integrated Woz Machine (IWM), 22
 Internal registers, 65816, 13-19
 Interrupt(s)
 control instructions, 129
 defined, 12-13
 disable (I) flag, 19

K

K (abbreviation for 1024), 4
 K/PC on debugger screen, 65
 KEEP option for ASML command, 184
 KEEP option for LINK command, 56
 KEY (keystroke modifier) on
 debugger screen, 64
 Key commands for menu items, 334
 Key-down event, 274
 Keyboard events, 274, 338-340

L

L (language card bank) on debugger
 screen, 65
 Label field, 33
 Labels
 characters in, 33
 global, 38
 local, 38
 Language card bank (L on debugger
 screen), 65

Line Editor (LineEdit), 163, 366
 Lines, drawing, 210-211
 LINK command, 56
 Linker, 29
 List Manager, 162
 Listing, columns in a, 60-61
 Listing directives, 41-42, 141
 Load and store instructions, 94-96
 Load and store macros, 146, 149
 Loading RAM-based tools from disk, 434
 Local labels, 38
 LocInfo data structure, 243-244
 Logical instructions, 119-122
 Logical operators, 45-46
 Longword (4-byte value), 168
 Low-resolution graphics mode, 191

M

M (machine state register) on debugger screen, 65
 M (memory/accumulator select bit), 19
 MACGEN (Macro Library Generator) utility, 142-143, 154
 Macintosh bit maps, 238-240
 Macro, defined, 133
 Macro directives, 136-141
 Macro language directives, 136, 138
 Macro libraries, creating, 142-143
 Macro libraries, updating, 143
 Macro library directives, 139
 Macros
 compared with subroutines, 133
 contents of, 134-135

 on the Programmer's Workshop Disk, 143-154
 using predefined, 154
 Managers in Apple IIGS Toolbox, 158-162
 Mask, pen, 201-202
 Master color value, 205
 Master pointer, 157
 Master scan line control byte, 196-197, 267
 Mega II chip, 20
 Megabyte, 10
 Memory
 banks, 11
 commands, debugger, 68-70
 in Apple IIGS, 22-25
 map, Apple IIGS, 22, 24
 organization, 11-12
 protection subdisplay, debugger, 68
 shadowing, 23
 Memory Manager, 156-157, 169
 starting the, 169
 Memory/accumulator select bit (M), 19
 Mensch, William D., Jr., 131
 Menu
 bar, 326-329
 events, 337-340
 items, 329-331
 modifiers, 334-337
 program (MENUS), 340-341
 Menu Manager, 158, 326
 Menu/item line list, 332-334
 Message, event, 279
 Messages, displaying, 232, 236-237
 Microsecond, 12

- Minimum version numbers of tool sets, 172-173
 - Minipalettes (640 mode color table groups), 206
 - Miscellaneous Tools tool set, 164, 169, 259-261
 - starting, 169
 - Mnemonic (opcode) field, 34
 - Mnemonics for 65816 instructions, 91-93
 - Modal
 - dialog boxes, 363
 - dialog box program (MODALD), 378
 - events, 375
 - programs, 271-272
 - Mode
 - control bits, 17
 - control instructions, 127-128
 - directives, 42
 - macros, 147, 153
 - Model program, listing for, 178-182
 - Modeless dialog boxes, 363-364
 - Modeless events, 376
 - Modifiers in event record, 279-281
 - Modifiers, menu, 334-337
 - Monitor, Apple IIGS, 25-26
 - Mouse events, 273, 337-338
 - Mouse pointer location, 279
 - Mouse pointer tool call
 - (ShowCursor), 232
 - Mouse-down event, 273
 - Mouse-reading tool calls, 286
 - Mouse-up event, 273
 - MouseWrite word processor, 29
 - MOVE file command, 57
 - Move macros, 146, 151-152
 - Multiprecision numbers, adding, 101
 - Multiprecision numbers,
 - subtracting, 103
 - Multisegment programs, 63
- N**
- N (negative) flag, 19
 - Nanosecond, 12
 - Native mode, 13, 17-19, 127
 - status bits in, 17-19
 - switching to, 127
 - NDA's — See *New desk accessories*
 - Negative (N) flag, 19
 - New desk accessories (NDAs), 162, 340
 - Nibble (4-bit number), 199
 - Note alert box, 365
 - Note Synthesizer, 162
 - Null event, 275-276
 - Numbers
 - arranging in increasing order, 110
 - binary-coded decimal (BCD), 99-101, 103
 - shifting signed, 126
 - signed, 99, 104
 - unsigned, 99
 - Numeric values, data formats for, 99-100
- O**
- Object module, 29
 - Opcode (mnemonic) field, 34-35
 - Operand addressing modes,
 - summary of, 74

Operand field, 34-35
 Operand formats for addressing modes, 86
 Operating modes, 13
 Operators, 42-47
 ORCA/M assembler, 27
 Ovals, drawing, 223-224
 Overflow, 19

P

P (Processor Status Register), 17-19
 Packed bytes of BCD digits, 100
 Page (256-byte memory unit), 11
 Page, direct, 11
 Page, zero (in 6502), 11
 Pattern, pen, 198-200
 PBR (Program Bank Register), 16
 PC (Program Counter) register, 16
 Pen, QuickDraw, 198-203
 attributes, 198
 location (PenLoc), 198
 mask, 201-202
 mode (PenMode), 198
 pattern (PenPat), 198-200
 size (PenSize), 198
 state, 198, 202-203
 Pencil display program (PENCIL), 249-250
 Pixel (picture element), 194-195
 Pixel image, 238-242, 247-249, 254
 Point in Quickdraw's drawing space, 194-195
 Pointer (4-byte address), 156, 168
 POLY (triangle-drawing program), 219-220

Polygons, drawing, 218-223
 PortRect, Quickdraw, 245
 Predefined controls, 357-359
 PREFIX command, 57
 PrepareToDie subroutine, 169
 Print Manager, 162
 Printer interface cards, 59-60
 Priorities, event, 276-277
 Processor status bit instructions, 123
 Processor Status Register (P), 17-19
 ProDOS 16 macros, 144-145
 Program Bank Register (PBR), 16
 Program control directives, 36-38
 Program Counter (PC), 16
 Program counter relative addressing modes, 83-84
 Program identification number, 169
 Program model, listing for, 178-182
 Programmer's Workshop —
 See Apple II GS Programmer's Workshop
 Programming errors, common, 189-190
 Programs, copying between, 186
 Pull-down menus, 328
 Push and pull instructions, 115-119
 Push and pull macros, 144-149

Q

Q (quagmire register) on debugger screen, 65
 Queue, event, 160, 271, 276-277
 QuickDraw II
 conceptual drawing space, 192-194
 described, 157-158

drawing environment, 192-196
starting and stopping, 196-197

R

Radio buttons, 159, 358
RAM subdisplay, debugger, 65-66
RAM-based tools, 171-176, 434
 loading from disk, 434
Read-Modify-Write instructions,
 86-88
Rectangle(s)
 drawing program (RECTS), 214
 drawing, 211-218
 frame (boundary) of, 211
 round-cornered (roundrects), 223
Regions, drawing, 224-225
Register commands, debugger,
 68-70
Register subdisplay, debugger,
 64-65
Register transfer instructions, 96
Registers, 65816, 13-19
Relational operators, 46-47
Relocatable blocks in memory, 156
RENAME file command, 57
Repeat count for DC directives, 40
Request, interrupt, 13
Results, reserving stack space for,
 167
ROM, Apple IIGS, 25
ROM-based tool sets, 171
Rotate instructions, 125
Roundrects (round-cornered
 rectangles), 223

S

S register (Stack Pointer), 15-16
S16 (ProDOS 16 system load) files,
 62
SANE (Standard Apple Numerics
 Environment), 163
Scan line control byte (SCB),
 196-197, 266-267
Scrap Manager, 162
ScreenMode global equate, 197
Scroll bars, 159, 297-298, 302,
 358-359
Serial Communications Controller
 (SCC), 22
Shadowing, memory, 23
Shapes, animating, 254
Shapes, calculation calls for,
 225-226
Shell load (EXE) file, 29
Shift and rotate instructions, 124-126
Shift macros, 146, 152-153
Shifting signed numbers, 126
SHOW command, 57
SHOWTEXT (text display program),
 286-287
Shutting down tool sets, 176-177,
 188-189, 434
Sign bit, 5
Signed arithmetic, 104
Signed numbers, 5, 99, 104, 126
 shifting, 126
Single-step commands, debugger,
 66-67
Software interrupts, 130
Sound Manager, 162

- Source program, 27
 - Source statements, 32
 - Space allocation directives, 39-40
 - Speaker-beeping program (BEEP), 48-49
 - Stack
 - addressing modes, 84
 - described, 15-16
 - instructions, 113, 115-119
 - space, reserving for a result, 167
 - subdisplay, debugger, 65
 - Stack Pointer (S), 15-16
 - Stack relative addressing modes, 84-85
 - Stages of an alert, 366
 - Standard color table
 - 320 mode, 205
 - 640 mode, 206
 - Standard File Operations tool set, 163
 - Start-up sequence for tool sets, 186
 - Starting ROM-based tool sets, 171
 - Status bit instructions, 123
 - Status Register, Processor, 17-19
 - flags in, 17
 - Stop alert box, 365
 - Store instructions, 96
 - Store macros, 146, 149
 - STR (string) macro, 150
 - Subdisplays on the debugger screen, 64-66
 - Subroutine instructions, 111-113
 - Subroutines compared with macros, 133
 - Subtraction, 103
 - Subtraction macros, 146, 149
 - Super high-resolution graphics modes, Apple IIGS, 192
 - Switch (applications) events, 275
 - Switching between native and emulation mode, 127-128
 - Symbolic parameter directives, 139-140
 - System load (S16) file, 29
 - System menu bar, 327
- T**
- Task
 - code, 309-310
 - mask, 308-309
 - record (TaskMaster event record), 308-309
 - TaskMaster, 307-311
 - Text display program (SHOWTEXT), 286-287
 - Text, displaying, 232, 236-237
 - Text Tools tool set, 163
 - Three-way decision sequence, 110-111
 - Tick, 279
 - Time and date operations, 254, 259-266
 - Title bar in a window, 295, 302
 - Tool calls, making, 166-168
 - Tool calls, using in programs, 165-166
 - Tool Locator, 156, 169
 - starting the, 169
 - Tool set interactions, 164-165, 435
 - Tool set numbers, 172, 436

Tool sets

- minimum version numbers of, 172-173
- shutting down, 176-177, 188-189, 434
- start-up sequence for, 186
- working space for, 169-171, 436-437

Tool table, 171

Toolbox — See *Apple IIGS Toolbox*

Top-down program design, 31-32

Trace commands, debugger, 66-67

Triangle-drawing program (POLY), 219-220

Two's-complement numbers, 6-7

TYPE command, 55

U

Unconditional transfer instructions, 106-107

Underscore prefix for tool calls, 166

Unsigned numbers, 99

Update (window) event, 274-275

Updating macro libraries, 143

User-defined events, 275

Utility macros, 144-154

V

V (overflow) flag, 19

Vectors, interrupt, 13, 128

Video Generator Chip (VGC), 22

W

Weights of bit positions, 2-3

Western Design Center, 10

Window

alert, 294, 295

components, 295-299

content region of a, 295

definition of, 294

display program (WINDOWS), 311-313

document, 294-295

events, 274-275

Window Manager, 158, 294

WINDOWS (window display program), 311-313

Word (2-byte value), 168

Working space for tool sets, 169-171, 436-437

Write macros, 147, 153-154

X

X (index register select) bit, 19

X register, 15

Y

Y register, 15

Z

Zero (Z) flag, 18

Zero page (in 6502), 11

Zoom box, 296, 302

Quick Index

Instructions

ADC, 100-101, 402
AND, 119-120, 403
ASL, 124, 403
BCC, 106, 108, 404
BCS, 106, 108, 404
BEQ, 106, 108, 404
BGE, 108, 404
BGE, 93, 404
BIT, 120, 122, 405
BLT, 93, 108, 404
BMI, 106, 108, 405
BNE, 106, 108, 406
BPL, 106, 108, 406
BRA, 106-107, 406
BRK, 130, 407
BRL, 106-107, 407
BVC, 106, 108, 407
BVS, 106, 108, 408
CLC, 100-101, 408
CLD, 100, 408
CLI, 129-130, 408
CLV, 100, 104, 409
CMA, 93, 409
CMP, 100, 105, 109-111, 409
COP, 130, 410
CPX, 100, 105, 109-111, 410
CPY, 100, 105, 109-111, 410
DEA, 93, 411
DEC, 100, 104, 411
DEX, 100, 104, 411
DEY, 100, 104, 411
EOR, 120-121, 412
INA, 93, 412
INC, 100, 104, 412
INX, 100, 104, 413
INY, 100, 104, 413
JML, 106-107, 413
JMP, 106, 414
JSL, 111, 113, 414
JSR, 111-112, 414
LDA, 94-95, 415
LDX, 94-95, 415
LDY, 94-95, 416
LSR, 124, 416
MVN, 95-99, 417
MVP, 95-99, 417
NOP, 131, 417
ORA, 120-121, 418
PEA, 116, 118, 418
PEI, 116, 118, 419
PER, 116, 119, 419
PHA, 115-117, 419
PHB, 115-116, 419
PHD, 115-116, 420
PHK, 115-116, 420
PHP, 115-116, 420

PHX, 115-116, 420
 PHY, 115-116, 421
 PLA, 115-116, 421
 PLB, 115-116, 421
 PLD, 115-116, 421
 PLP, 115-116, 422
 PLX, 115-116, 422
 PLY, 115-116, 422
 REP, 120, 123, 422
 ROL, 124-125, 423
 ROR, 124-125, 423
 RTI, 129-130, 424
 RTL, 111, 113, 424
 RTS, 111-112, 424
 SBC, 100, 103, 425
 SEC, 100, 103, 425
 SED, 100-101, 103, 426
 SEI, 129-130, 426
 SEP, 120, 123, 426
 STA, 95-96, 427
 STP, 131, 427
 STX, 95-96, 428
 STY, 95-96, 428
 STZ, 95-96, 428
 SWA, 93, 433
 TAD, 93, 429
 TAS, 93, 430
 TAX, 95-97, 429
 TAY, 95-97, 429
 TCD, 95-97, 429
 TCS, 95-97, 430
 TDA, 93, 430
 TDC, 95-97, 430
 TRB, 120, 122, 430
 TSA, 93, 431
 TSB, 120, 122, 431
 TSX, 95-97, 431

TXA, 95-97, 432
 TXS, 95-97, 432
 TXY, 95-97, 432
 TYA, 95-97, 432
 TYX, 95-97, 433
 WAI, 130-131, 433
 WDM, 131
 XBA, 95-97, 433
 XCE, 127, 433

Assembler Directives

65816, 37, 42
 65C02, 37, 42
 ABSADDR, 37, 41
 ALIGN, 43
 APPEND, 37, 39
 CASE, 43
 COPY, 37, 39
 DATA, 37-38
 DC, 37, 39
 DS, 37, 39
 EJECT, 37, 42
 END, 36-37
 ENTRY, 37-38
 EQU, 37, 40
 EXPAND, 43
 GEN, 138, 141
 GEQU, 37, 40
 IEEE, 43
 INSTIME, 43
 KEEP, 37, 39
 LIST, 37, 41
 LONGA, 37-38
 LONGI, 37-38
 MACRO, 134, 137

MCOPY, 137, 139, 165
MDROP, 137, 139
MEM, 43
MEND, 135, 137
MERR, 43
MLOAD, 137, 139
MSB, 43
OBJCASE, 43
ORG, 37, 40
PRINTER, 37, 41
PRIVATE, 43
PRIVDATA, 43
RENAME, 43
SETCOM, 43
START, 36-37
SYMBOL, 37, 41
TITLE, 37, 42
TRACE, 139, 141
USING, 37-38

Tool Calls

Alert, 393-394
Button, 282, 286
CautionAlert, 365, 394-396
ClearScreen, 210
CloseDialog, 374-375
ClosePoly, 218-219
CloseRgn, 225-226
CloseWindow, 301, 307
CopyRgn, 226
CtlShutDown, 312, 361
CtlStartup, 312, 361
DialogSelect, 375-376
DialogShutDown, 367
DialogStartup, 367

DisposeAll, 188
DisposeHandle, 188
DoKeyDown, 277
DoMouseDown, 277
DrawChar, 237
DrawCString, 237
DrawMenuBar, 333
DrawString, 237
EMShutDown, 281
EMStartup, 281-283
EraseOval, 224
ErasePoly, 219
EraseRect, 212
EraseRgn, 226
FillOval, 224
FillPoly, 219
FillRect, 212
FillRgn, 226
FixAppleMenu, 340
FixMenuBar, 333
FlushEvents, 282, 286
FrameOval, 224
FramePoly, 219
FrameRect, 211-212
FrameRgn, 226
GetBackPat, 210
GetColorTable, 267, 269
GetMouse, 282, 286
GetNextEvent, 281, 283-285, 307
GetPen, 203
GetPenMask, 204
GetPenPat, 204
GetPenSize, 204
GetPenState, 203
GetPort, 247
GetPortRect, 248
GetPrefix, 176

GetText, 375-377	PaintPoly, 219
GetTick, 266	PaintRect, 212
HideWindow, 302, 307	PaintRgn, 226
InitColorTable, 267, 269	PenNormal, 202-203
InsertMenu, 333	PenPat, 208
IsDialogEvent, 375-376	PPToPort, 247-249
LEShutDown, 367	QDShutDown, 197
LEStartup, 367	QDStartup, 196
Line, 210-211	Quit, 177
LineTo, 210-211	ReadAsciiTime, 259-260
LoadOneTool, 187, 225	ReadTimeHex, 259-260
LoadTools, 171, 187	Refresh, 301
MenuSelect, 337	RemoveDItem, 374
MenuShutDown, 333	ScrollRect, 248
MenuStartup, 333	SelectWindow, 302, 307
MMShutDown, 187	SetBackColor, 237
MMStartup, 169, 187	SetBackPat, 210
ModalDialog, 374-375	SetColorEntry, 267
Move, 203	SetColorTable, 267, 269
MoveTo, 203, 211	SetForeColor, 237
MTShutDown, 188, 260	SetMasterSCB, 267, 269
MTStartup, 169, 188, 260	SetOrigin, 248-249
NewDItem, 374	SetPenMask, 204
NewHandle, 169-170, 187	SetPenPat, 204
NewMenu, 333	SetPenSize, 204
NewModalDialog, 373	SetPenState, 203
NewModelessDialog, 374	SetPort, 247
NewRgn, 225-226	SetPortRect, 248-249
NewWindow, 300-306	SetPrefix, 176
NoteAlert, 365, 394-396	SetRect, 226
OffsetPoly, 226	SetRectRgn, 226
OffsetRect, 226	SetSolidBackPat, 208
OffsetRgn, 226	SetSolidPenPat, 208, 210
OpenPoly, 218-219	ShowCursor, 232
OpenRgn, 224, 226	ShowWindow, 301, 306
PaintOval, 224	SolidPattern, 210

StopAlert, 365, 394-396

SysFailMgr, 188

TaskMaster, 307-311

TLMountVolume, 176, 187

TLShutDown, 187

TLStartup, 169, 187

WindShutDown, 301

WindStartup, 301

Special Software Offer!

Get a running start on IIGS software development with all the sample programs from this book!

The software featured in Apple IIGS Assembly Language Programming is available for only \$9.95, plus \$3.00 for shipping and handling. Why spend several hours typing it in yourself? Send us your check, or call our 800 number, and we'll send you a 3.5" IIGS disk with all the source code, macros, exec files, and executable load files from the sample programs in the book.

This disk is not available in stores; it can only be purchased directly from the publisher.

Use this coupon to order or call 1-800 223-6834, ext. 479, and have your credit card information handy.

Mail to: Bantam Books, Inc. Dept. IIGS
666 Fifth Avenue
New York, NY 10103

Yes! Send me the Apple IIGS Assembly Language Program Disk (50049-X) for only \$12.95 (\$9.95 plus \$3.00 for UPS shipping and handling).

Name _____

Address _____

City _____ State _____ Zip _____

☐ My check or money order for \$12.95 is enclosed.
(Please make check payable to Bantam Books, Inc.)

☐ Charge my _____ Visa _____ MasterCard _____ American Express

Acct. # _____ Exp. Date _____

Signature _____

Please allow 3-4 weeks for delivery. Price shown in U.S. dollars. Price and availability subject to change without notice. This offer is not available in stores. No C.O.D.s.

IIGS—8/87